MIT/LCS/TR-241

REPRESENTATION AND ANALYSIS OF REAL-TIME CONTROL STRUCTURES

Rowland F. Archer, Jr.

Auaust 1980

REPRESENTATION AND ANALYSIS OF REAL-TIME CONTROL STRUCTURES

ROWLAND FRANK ARCHER, JR.

September, 1978

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE

CAMBRIDGE MASSACHUSETTS 02139

REPRESENTATION AND ANALYSIS OF REAL-TIME CONTROL STRUCTURES

by
ROWLAND FRANK ARCHER, JR.

Submitted to the Department of Electrical Engineering
and Computer Science
on August 18, 1978 in partial fulfillment of the requirements
for the Degree of Master of Science

ABSTRACT

A new notation is introduced for representing real-time scheduling at the task and event level. These schedules are called control structures. The primary constructs included which direct the flow of control are sequencing, iteration, and preemption. Additional notation allows the representation of interrupt masking, task termination by external events, task restart as well as resumption from the point of preemption and codestripping. Algorithms are given for finding the preemption structure of a given control structure in the notation.

The types of representable control structures are classified by the topology of their Control Flow Graphs. It is shown that although branching is allowed in the preemption structure, a tree-shaped preemption structure cannot be represented. Both partial and total orderings of tasks and interrupt priorities are supported, however.

A terminology for describing real-time properties of control structures is developed, and it is seen that without certain assumptions about task execution times and event timings, conclusions cannot be drawn regarding real-time performance of a control structure. A series of algorithms is presented which make use of these assumptions, and find values for task execution times in the presence of preemption. The algorithms can analyze control structures containing the principal control features; suggestions are given for further development of algorithms which can analyze any representable control structure.

Thesis Supervisor: Stephen A. Ward
Assistant Professor of Electrical
Engineering and Computer Science

Key words and phrases: Real-time, control structure,
control flow graph, scheduling, interrupts, latency, codestripping.

Acknowledgements

Primary thanks are due to Steve Ward for the germinal idea, and his guidance in helping me to develop it. His resourcefulness was responsible time and again for keeping this research in motion.

Tom Teixeira's work in this area was also invaluable, especially for his careful and rigorous definitions of real-time properties of control structures.

The excellent systems programming support of the DSSR group has provided an exceptionally hospitable environment in which to program and produce this document.

My wife Lizbeth deserves mention for her encouragement and for doing more than her share so that my attention could stay focused on this investigation.

TABLE OF CONTENTS

LIST OF FIGURES

1: Introduction

In an article entitled "Toward a discipline of real-time programming" [Wirth 77b], Niklaus Wirth has divided programming into three categories based on the increasing complexity of validating their programs:

1. Sequential programming
2. Multiprogramming
3. Real-time programming

In a real-time system, a program may be attempting to control or to react to certain external processes which cannot be forced to cooperate with programmed processes through use of a synchronization primitive such as a semaphore [Dijkstra 68] or a monitor [Hoare 74]. In order to coordinate itself with these external, non-programmable processes, the real-time program must know something about its own execution speed. Thus its correctness will be dependent on the speed of the processor on which it is run; but this is not a property of the program itself; Wirth identifies this as the essential distinguishing feature of real-time programming.

This thesis does not directly address the issue of validating real-time programs. Instead, it deals with the representation of schedules for real-time programs called *control structures*, and some aspects of measuring real-time properties of the resulting control structures. In the sense that knowledge of these real-time properties may be a prerequisite for validation of a real-time program, the work presented here does represent a contribution to one aspect of the validation problem.

1.1: Related Research

Most of the previous studies in the field of real-time programming have been centered on one of two major areas, the design of languages for real-time programming, and scheduling to meet real-time deadlines.

The development of languages for real-time programming can be split between two approaches; the extension of existing languages [Benson 67; Freiburghouse 77; Ormicki 77; Phillips 76; Wirth 77a], and the creation of entirely new languages tailored to the requirements of real-time programs [Hennessy 75; Kieburtz 75; Schoeffler 70]. The essence of these efforts has been to provide some interface between the real-time program and the scheduling of itself and other programs, either through access to the processor's interrupt system, clocks and/or timers, or by influencing the processor's scheduling routines. Such features provide only a low level capability for determining a process' real-time behavior; in some cases it may be possible to think of all the timing interactions that could impact on the correctness of a real-time system, but the burden of doing so has usually fallen most heavily (and often totally) on the programmer. Decisions as basic as assigning priorities to different tasks must typically be made by manual analysis, in the hope that nothing has been overlooked. As the size of the system increases, the complexity of the problem grows as well, until manual analysis becomes extremely tedious and error-prone, if not impossible.

Ideally, a programmer could submit his real-time response requirements along with his programs, and either have them scheduled appropriately, or be told that his requirements cannot be met by that particular system. Some systems (such as the CONSORT system of the Domain Specific Systems Research group at MIT) have been developed which can do this for a limited class of programs, but to the author's knowledge no one has yet created a system to do this in general. However, considerable research has been done on scheduling tasks in the presence of hard real-time deadlines.

Most of the significant results obtained have been based on restricting attention to limited classes of control structure types. For example, in a multiprocessor environment where there is a partial ordering of tasks but no iteration outside of tasks, [Manacher 67] gives an algorithm which will construct near-optimal task lists (execution orderings) for almost any combination of task run times and deadlines. If the schedule is full to capacity with tasks whose completion times are guaranteed, his strategy allows the system to take on additional unguaranteed tasks without affecting the guaranteed status of those tasks already scheduled. His scheme does not consider the effects of preemption, however. Serlin [Serlin 72] and Liu and Layland [Liu 73] have independently studied the problem of scheduling tasks which are iterated but have no relative orderings. Serlin gives scheduling algorithms based on fixed priorities, time slicing, and relative urgency. The last is a dynamic priority scheme, where the processor re-evaluates the priorities of each task at every interrupt, and selects for execution the one with the earliest deadline. This method is shown to produce a schedule which meets real-time deadlines if any schedule will, but Serlin's analysis neglects the overhead of context switching.

A different approach is taken by [Hennessy 75; Kleburtz 75] in their microprocessor language TOMAL; instead of using an interrupt system, they have a compiler insert calls to a task control monitor (which is created along with the compilation of a set of programs) at specific points in the compiled code. This provides assurance that the task control monitor will get control within a finite and bounded interval, after each *codestrip*, as the code between monitor calls is named. This is similar to a time slicing system which allocates execution time in fixed amounts to each task, but the time slices are synchronized with program execution. The length of the codestrip is determined by the response time requirements of the task, and the compiler can determine whether the programmer supplied requirements are in conflict. The notation given in this thesis has the capability of representing codestrips.

A work which is related to the present one and in fact complementary is that of Teixeira [Teixeira 78]. Much of the terminology used here was developed there, particularly that of Chapter 5, where algorithms for measuring real-time properties are developed. Teixeira also used the regular expression notation of Chapter 2 to denote sequencing and iteration of tasks. His study centers, however, on finding schedules to meet real-time constraints; the orientation of the present work is described in the following section.

1.2: Objectives

The principal goal of this research is twofold; to develop a convenient representation for real-time control structures, and to demonstrate how such a representation is useful as a basis for analyzing real-time properties for specific control structures.

The representation as developed models control structures at the task and interrupt level; the tasks are assumed to be self-contained program units whose execution time is bounded, and interrupts are represented as occurrences of event variables. The event variables could be used to represent any event however, which might be synchronous or asynchronous with respect to the executing task. The notation can represent total and partial orderings among its tasks, and iteration of tasks at a single priority level or across several priority levels. As well as representing conventional single and multi-level interrupt structures, the control structures given here can represent several unconventional preemption structures, including branched structures where each branch has an individual preemption structure which may itself be branched.

As well as representing this basic framework, the capability is provided to represent:

1. Codestripping as previously described.

2. External termination of a task or group of tasks by an event occurrence (as opposed to temporarily preempting them).

3. Indication that a task or group of tasks is not preemptible by a set of events.

4. The choice between restarting a preempted task or group of tasks from the beginning vs. resuming execution from the point of interruption.

Thus a rather general notation is given, which in addition represents all of this information rather compactly. The notation may be used in any application where it is necessary to communicate something about a control structure of this sort, be it human to human, human to machine, or machine to machine. In the second case, the specific applications in mind are representation of a control structure for analysis, and for describing to a real-time system what sort of control structure it should establish for a set of tasks with real-time constraints. In this vein, the notation is quite independent of machine architecture, and thus a subset of the language can be chosen for a target machine which supports the control features included therein.

This leads into the second goal of the investigation, which is to demonstrate how algorithms can be developed which ascertain real-time properties for control structures of the language. There are several time intervals which are probably of common interest to a large segment of users of real-time programs, such as:

1. The maximum delay between the occurrence of an event and the initiation of its program.

2. The maximum time required to execute a set of tasks at a given priority, with preemption.

3. The maximum time that may elapse without there being an execution of a given set of tasks.

This is not intended to be an exhaustive survey of real-time properties, but rather an introduction to the usage of the notation as the foundation for such analysis. Indeed, it is likely that each real-time system has its own special requirements and characteristics; it is hoped that an appropriate subset of the language can be chosen to model those characteristics, and algorithms developed which are suited to an application's special needs. In addition, many applications will have natural restrictions which lead to simpler algorithms; it is with intent of illustrating this point that several special case algorithms are developed.

1.3: Outline of the Thesis

The next chapter presents a context free grammar for the control structure language, as well as giving the semantics for each construct. Sequencing, iteration and preemption are the principal features, with extensions added as described in Section 1.2. Methods of determining the overall preemption structure of a control structure are also presented.

Having introduced the notation, Chapter 3 presents the concept of a Control Flow Graph (CFG) [Allen 76; Fosdick 76], which gives a graphic representation for the paths of control flow dictated by a given control structure. A definition of absolute priority levels is derived from a control structure's CFG representation. Then a classification of control structure types representable by the notation is given, based on the topology of their CFG's. In addition, some types of control structures which are *not* representable are described.

A terminology for real-time properties of control structures is developed in Chapter 4; the requirements for knowing certain things about event timings in advance is also discussed here.

This leads into Chapter 5, where a hierarchical series of algorithms is presented which are designed to find the worst cases for some of the real-time properties of increasingly complicated classes of control structures. The most general algorithm given is applicable to the set of control structures which includes the basic framework of sequencing, iteration and preemption. The types of modifications which would be required to analyze any representable control structure are discussed, although detailed algorithms are not given.

2: A Notation for Real-time Control Structures

2.1: Introduction

In this chapter a notation for representing real-time control structures will be developed. The intention is to provide a general analytical tool which will be suitable for representing most of the possible ways to share a processor among the members of a set of tasks. This will include:

1. Sequencing: a total ordering of tasks to be executed.
2. Iteration: cyclic execution of some ordered set of tasks.
3. Preemption: a partial ordering of tasks where the occurrence of an *event* forces termination of execution of the currently running task and starts execution of a new task.

A context-free grammar will be developed to define the syntax of the representation. It is summarized in Appendix A.

2.2: The Basic Control Structure

The real-time system to be represented is modelled as a set of procedures to be run, called *tasks*, a *control structure* which specifies the order (or possible orders) in which the tasks may be run, and a *processor* which executes the tasks according to the scheduling constraints specified by the control structure.

Thus the flow of data between tasks, if there is any, need not be a concern; it is assumed that any execution ordering needed to preserve the intended semantics of the computation (data flow) will be embodied in the control structure. For example, if an output of task A is an input of task B, then the control structure associated with their execution should ensure that task A completes execution before task B begins.

Further, the detailed flow of information and control *within* a task, i.e. among its internal variables and instructions respectively, need not be of concern either. It is only necessary that an upper bound on the execution time of a task be established; this is discussed further in Section 4.2.

A task will be represented by a *task identifier* ("<task id>"), which in most of the examples will be a single capital letter (though it need not be). Figure 2.1 shows the grammar which defines task identifiers.

```
<task id> ::= <letter> | <task id> <alphanumeric>
<letter> ::= A | B | C | ... | Z
<alphanumeric> ::= <letter> | <digit>
<digit> ::= 0 | 1 | 2 | ... | 9
```

Fig. 2.1. Syntax for task identifiers.

Next to a single task, the simplest thing to represent is the sequencing of two or more tasks which are totally ordered. This is done in the natural way, by listing the task identifiers in the order of execution of their corresponding tasks, separated by blank spaces for parsing. A string of one or more tasks will be called a *basic control structure*, or <basic cs>. Note that it is permissible to list a task id more than once in a <basic cs>, to represent the situation where the corresponding task is executed more than once with zero or more other task executions sandwiched in between.[1] The syntax is:

> <basic cs> ::= <task id> | <basic cs> ƀ <task id>

where "ƀ" represents the blank space terminal symbol.

The simplest control structure is just a basic control structure:

> <control structure> ::= <basic cs>

Thus the grammar given so far is sufficient to represent single task execution and sequenced task execution control structures.

2.3: Flow of Control

It is useful to formalize the notion of control flow with respect to control structure execution. The processor follows the "instructions" supplied by a control structure, doing both "applications-oriented" work (when it is actually executing the statements of a task), and "systems-oriented" work (when it is determining which task to execute next according to the constraints embedded in the control structure). In either case, the actual machine instructions being executed at any time will be associated with a particular symbol in the control structure representation; it will be said that at that time the *locus of control* (abbreviated LC) is at that symbol. For example, in the following control structure:

[1] Every occurrence of a task id in a control structure represents a separate instantiation of that task, with its own private state. This is used to model reentrantly coded routines.

A B

when instructions of task A are executing, LC is at A; when instructions of task B are executing, LC is at B.

2.4: Closed Control Structures

It is desirable to introduce parenthesization for the grouping of task id's in the natural way. In particular, this will be needed to indicate the *scope* of the various special symbols which will be used for iteration, preemption, etc. It will also be helpful in constraining the class of legal control structures to exclude nonsensical ones, such as those in which some tasks can never execute, regardless of preemption timing considerations. Parenthesized (sub-)control structures will be called *closed control structures*, and the class will be added to as necessary for additional representational power. At the top level, closed control structures will be included in the set of legal control structures. Figure 2.2 gives the syntax for closed control structures; a syntax is also given for closed control structure lists, which will be needed later to represent more complex control structures.

```
<control structure> ::= <basic cs> | <closed cs>
<closed cs> ::= (<basic cs>) | (<closed cs> <basic cs>) | (<closed cs list>)
<closed cs list> ::= <closed cs> | <closed cs list> <closed cs>
```

Fig. 2.2. Syntax for closed control structures.

2.5: Iteration

Most real-time process control applications require the periodic repetition of a certain task or sequence of tasks. Borrowing from the notation of regular expressions, the asterisk is used to indicate a endless repetition of a control structure. Its BNF:

<iterative cs> ::= <basic cs>* | <closed cs>* | <basic cs> <iterative cs>

The use of "*" is most easily explained by examples:

$$A\ B^* \equiv A(B)^* \equiv A\ B\ B\ B\ B\ B\ \ldots$$

$$(A\ B)^* \equiv A\ B\ A\ B\ A\ B\ \ldots$$

From a flow of control viewpoint, when LC reaches an asterisk following a right parenthesis, it returns to the matching left parenthesis. If it reaches an asterisk following a task id, it repeats that task.

The final expansion of the top-level definition of control structure is:

<control structure> ::= <basic cs> | <closed cs> | <iterative cs>

2.6: Preemption

2.6.1: Preemptible Control Structures

With the class of control structures defined so far, the only execution sequences possible are those in which the order of task execution is entirely predetermined (static). In many situations, a processor will need to respond to asynchronous events such as interrupts, which may not occur at predictable times. It may be desirable to have such events trigger the execution of a different part of the control structure than was previously in control. Informally, this will be modelled by placing sub-control structures into the overall control structure in order of non-decreasing priority. Demarcation of the priority levels is achieved by indicating that a control structure is *preemptible*. Figure 2.3 gives the syntax for preemptible control structures. Preemption is initiated by occurrence of a particular event (which may be complex),[1] so an *event variable* is included which stands for the event.

```
<preemptible cs> ::= <control structure> / <event var>

<event var> ::= e<integer>

<integer> ::= <digit> | <integer> <digit>

<closed cs> ::= (<basic cs>) | (<closed cs> <basic cs>) | (<closed cs list>) |
                      (<preemptible cs>) | (<closed cs> <preemptible cs>)
```

Fig. 2.3. Syntax for preemptible control structures.

Consider the following simple example, which models a control structure with a single level of interruption:

$$((A^*/e1)B)^*$$

The interpretation of this control structure is that A runs repeatedly until event e1

[1] The event variable itself is not complex, but it may *represent* a complex event.

happens; this initiates B, which executes once; then LC returns to A*.

The next section will describe how more complex control structures are represented (using the above syntax), such as those having multiple levels of interruption.

2.8.2: Multiple Priority Level Control Structures

Informally, event variables lie at the interface between control structures of different priority, the control structure to the left of the "/<event var>" construction having the lower priority. If LC is in the lower priority control structure when the event happens, it will move to the control structure immediately to the right of the event variable.

Thus a control structure with three priority levels might appear as:

(((((A B)*/e1)C D)/e2)E)*

The preemption structure (for each event, the tasks which it may preempt) is fairly straightforward here; e1 preempts A or B, e2 preempts A, B, C or D. But the notation is capable of representing more complex control structures, and a method of precisely determining the preemption structure is needed.

The "interrupts" or "preempts" relation is transitive; if e1 interrupts A, initiating C, and C is interruptible by e2, then A is interruptible by e2. Moreover, all tasks of a single basic control structure will run at the same priority level, so basic control structures can be considered as units, rather than examining the preemp-

[1] Although a later section will introduce the capability of *masking* specific inter-

tion of individual tasks.[1]

The "interrupts" relation will now be formalized, i.e. it will be established clearly for each event in a control structure which basic control structures it may preempt. The set of tasks which are interruptible by a certain event will be referred to as the *scope* of that event. The "interrupts" relation for a control structure will be represented by a Boolean matrix I with n rows and columns, where n is the number of basic control structures in the control structure being analyzed. A single basic cs is associated with each row i and column i, for $1 \le i \le n$. The basic cs associated with row (and column) i will be referred to as "basic cs i."

The first event to the left of each basic cs will be called that basic cs's *initiating event*. If I[i,j] = 1, it means that basic cs i runs at a higher priority than basic cs j; in particular, it means that basic cs i's initiating event can preempt basic cs j. The matrix I is computed according to the algorithm given in Figure 2.4.

This matrix specifies which events cause preemptions across the border between adjacent priority levels. Since the "interrupts" relation is transitive, the transitive closure of this initial relation is the complete preemption structure; this specifies, for each event in the control structure, exactly which basic cs's it can preempt. Computing the transitive closure of the relation represented by I is straightforward. Let I+ be the transitive closure of I. Then $I+ = I + I^2 + \ldots + I^n$, where + is normal matrix addition. Boolean matrix multiplication is performed like regular matrix multiplication except 'AND' is substituted for 'TIMES' and 'OR' for

rupts while a particular task is executing.

Algorithm 2.1:

1. Let n be the number of basic cs's in the control structure. Associate a unique integer from 1 to n with each basic cs.

2. Initialize I to be an $n \times n$ matrix of zeroes.

3. For each basic cs i, do steps 4 and 5.

4. If basic cs i has no initiating event, leave row i of I equal to all zeroes.

5. If basic cs i has an initiating event e, find the control structure immediately preceding the construction "/e." Call this "control structure k." By the syntax of preemptible control structures, control structure k will be either a basic, closed or iterative cs. For each basic cs j in control structure k, set I[i,j] equal to 1.

Fig. 2.4. Computing the matrix I.

'PLUS'.

Consider an example of a control structure which contains preemptible control structures, and which can be used to illustrate the construction of the "interrupts" relation:

Example 2.1. (((((A B)*/e1)C)*/e2)((D/e3)E))*

Notice that this control structure contains four basic control structures, A B, C, D and E. The initiating events for these basic cs's are as specified in Figure 2.5.

Basic CS	Initiating Event	Row/Column of I
A B	none	1
C	e1	2
D	e2	3
E	e3	4

Fig. 2.5. Initiating events for Example 2.1.

The matrix I is formed following Algorithm 2.1, and it appears in Figure 2.6.

1. A B has no initiating event, so row 1 = [0 0 0 0]

2. C's initiating event is e1. The control structure preceding e1 is (A B)*, which contains the basic cs A B. Thus I[2,1] := 1.

3. D's initiating event is e2. The control structure preceding e2 is ((A B)*/e1)C)* which contains the basic cs's A B and C. Thus I[3,1] := 1 and I[3,2] := 1.

4. E's initiating event is e3. The control structure preceding e3 is D. Thus I[4,3] := 1.

I	A B	C	D	E
A B	0	0	0	0
C	1	0	0	0
D	1	1	0	0
E	0	0	1	0

Fig. 2.6. The I matrix for Example 2.1.

Now, to get the overall preemption structure, compute I+, the transitive closure of I, as shown in Figure 2.7.

I+	A B	C	D	E
A B	0	0	0	0
C	1	0	0	0
D	1	1	0	0
E	1	1	1	0

Fig. 2.7. I+ for Example 2.1.

The preemption relations of the control structure are summarized in Figure 2.8.

LC at	Preemptible by	Initiates
A or B	e1	C
A or B	e2	D
A or B	e3	E
C	e2	D
C	e3	E
D	e3	E
E	none	none

Fig. 2.8. Preemption structure for Example 2.1.

2.6.3: Occurrence of Events

The notion of an event "happening" is purposefully left vague; each application of the notation can attach its own meaning. For the purpose at hand it is sufficient to assume that an event variable is like a flag,[1] which gets set when its associated event occurs. The processor checks all the event variables before beginning execution of every instruction. The following informally describes what happens if any flag is found to be set:

1. In the case where LC is to the right of the event variable which has been set, no immediate effect on execution of the currently running task results. The currently running task is of a higher priority than that which is requesting the interrupt.

2. The event variable remains set until such time as LC is to the left of it and in a basic cs which is preemptible by it, at which time it will cause a preemption.

3. If more than one event corresponding to event variables to the right of LC has happened, then the rightmost one represents the highest priority interrupt (request), and LC moves to the right

[1] Generally, a queue of requests is associated with a given event variable, so that additional occurrences of the event will be remembered if they occur before the initial occurrence is noted by the processor. By specifying a length for this queue, a system which remembers an arbitrary number of event occurrences can be modelled.

of it (assuming, of course, that LC was within a basic cs preemptible by the event).

4. Completion of the control structure at a given priority "resets" the event variable which triggered its execution; note that this must be done at completion rather than at initiation so that if the control structure is preempted before it completes, then LC will return to it when it is once again the highest priority control structure requesting processor service.

2.6.4: Substructure at a Single Priority Level

A useful extension to the scheme is to provide for arbitrarily many control structures[1] to reside at the same priority level, but to be initiated by different events. During execution of one of these control structures, occurrence of events in the other(s) at the same priority level will have no (preemptive) effect. The principle syntactic change is to allow replacement of an event variable by an *event coupled list*, as shown in Figure 2.9.

```
<preemptible cs> ::= <control structure> / <event list>

<event list> ::= <event var> | (<event coupled list>) |

                 (<event coupled list>)*

<event coupled list> ::= <event var>: <control structure> |

                 <event coupled list> '|' <event var>: <control structure>

where '|' means the terminal symbol |.
```

Fig. 2.9. Syntax for event coupled preemptible control structures.

[1] Of arbitrary complexity, e.g. there may be additional local priority structure.

Consider an example:

$$(A^*/(e1: B \mid e2: C))^*$$

Preemption rights are as follows:

LC at	Preemptible by	Initiates
A	e1	B
A	e2	C
B or C	none	none

Execution of B or C continues uninterrupted to termination. Termination of B or C returns LC to A (unless e1 or e2 has happened again).

A slight modification in the position of the terminal '*' leaves the interrupt structure the same but results in different behavior on termination of B or C:

$$(A^*/(e1: B \mid e2: C)^*)$$

The idea here is that once either B or C has been initiated (through occurrence of e1 or e2, respectively), control is never again returned to A. Instead, B and C will be executed every time e1 or e2 occurs.

2.6.5: Determining the Interrupt Structure

Since arbitrary control structures may reside in an event coupled list, it follows that such structures may contain additional events (or event coupled lists) which trigger even more deeply nested control structures.

This ability to nest control structures raises a new semantic issue; what should be the scope of events which are not at the top level in the event coupled list? The choice made here is to let *any* event in an event coupled list have the same scope external to the event coupled list that an event variable would have if it were substituted for the event coupled list. Consider the following:

Example 2.2. (A/(e1:((B/e2)C)|e3:((D/e4)E)))*

The scope of e1, e2, e3 and e4 external to the event coupled list (e1:((B/e2)C)|e3:((D/e4)E)) is the same as that of e5 in:

(A/e5)

namely, the control structure to the left of the slash in the construction "/(<event coupled list>)".

The initiating events, as shown in Figure 2.10, are determined as before: the first event variable to the left of each basic cs. The *internal* scope of the event variables is somewhat different, though. Events in event coupled lists may not preempt any task in the list which is separated from the event by a "|". Thus in the above example, e3 and e4 may not preempt B or C. Therefore Algorithm 2.1 must be modified to reflect this. Figure 2.11 shows the resulting algorithm.

Basic cs	initiating event
A	none
B	e1
C	e2
D	e3
E	e4

Fig. 2.10. Initiating events for Example 2.2.

Algorithm 2.2:

1. Let *n* be the number of basic cs's in the control structure under examination. Associate a unique integer from 1 to *n* with each basic cs.

2. Initialize I to be an *n*x*n* matrix of zeroes.

3. For each basic cs *i*, do steps 4 and 5.

4. If basic cs *i* has no initiating event, leave row *i* of I equal to all zeroes.

5. If basic cs *i* has an initiating event e, then this event appears in either a "/*e*" construction or a "|e" construction.

   a. If e appears in a "/*e*" construction, call the control structure immediately preceding "/*e*" "control structure *k*." For each basic cs *j* in control structure *k*, set I[*i*,*j*] equal to 1.

   b. If e appears in a "|e" construction, then e cannot preempt any other basic cs's in the event coupled list of which it is a member. Its scope starts at the control structure to the left of the "/" in the construction "/(<event coupled list>)". This will be the control structure preceding the first unmatched left parenthesis to the left of e. Call this "control structure *k*." For each basic cs *j* in control structure *k*, set I[*i*,*j*] equal to 1.

Fig. 2.11. Computing I for cs's containing event coupled lists.

The control structure of Example 2.2 has the following preemption relationships:

LC at	Preemptible by
A	e1, e2, e3, e4
B	e2
C	none
D	e4
E	none

Since two or more tasks may reside at the same priority level, such as B and C above, a natural question arises; what happens if both e1 and e3 occur "simultaneously," at least within the resolution of the interrupt system.[1] Most systems adopt some arbitrary metric to resolve such situations. A typical one is the distance of the interrupting device from the CPU. A similar approach is taken here. If more than one event is found to have occurred at the same priority level, then control is arbitrarily given to the *first* (leftmost) one in the event coupled list.

However, with the addition of event coupled lists, "forks" are introduced into the preemption structure, as shown in Figure 2.12. A diagram such as this is called a *Control Flow Graph*, and will be defined formally and used extensively in the next chapter. For now it is sufficient to note that this diagram "unravels" the preemption structure so that the relative priority levels of each task are displayed. If two or more events happen together, priority is given to the event which initiates the task having higher priority, as was done before. In the above example, if e1 and e4 happen simultaneously control is given first to E (which e4 initiates).

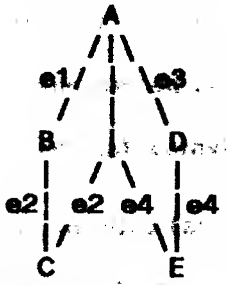

Fig. 2.12. Preemption structure for Example 2.2.

[1] Typically the presence of interrupt requests will be checked for once per instruction cycle, so any interrupts happening between two such checks will be indistinguishable as to their ordering in time.

2.7: Non-preemptible Tasks

It is occasionally necessary to perform all or some subset of a control structure's tasks in a non-preemptible mode, even though in the latter case other tasks at that priority level may be preemptible. Simply indicating that a task is non-preemptible is equivalent to saying that the interrupt system is "turned off" while that task is in execution. For generality, the notation allows as an alternative the specification of exactly those events which are not allowed to interrupt the task. Both capabilities are provided with the augmented syntax, shown in Figure 2.13. The scope of the symbol for non-preemptibility extends to closed control structures in the natural way, i.e. every task in the closed cs is non-preemptible.

```
<basic cs> ::= <task> | <basic cs> ɏ <task>

<task> ::= <task id> | <non-preemptible tid>

<non-preemptible tid> ::= '<task> | '(<ev list>)<task>

<ev list> ::= <event var> | <ev list>,<event var>

<non-preemptible closed cs> ::= '<closed cs> | '(<ev list>)<closed cs>

<closed cs> ::= ...(same as before plus: )... | <non-preemptible closed cs>
```

Fig. 2.13. Syntax for non-preemptible tasks.

Prefixing a task id (or a closed cs) with an apostrophe (e.g. 'A) indicates that that task is not preemptible by *any* event. If there is an event list after the apostrophe (e.g. '(e1)A), then that task is not preemptible by any event in the event list. Furthermore, it is not preemptible by any event which could lead to preemption by an event in the event list. For example:

(((((A*/e1)‘(e3)B C)/e2)D/e3)E)*

Here if LC is at B, it is not preemptible by e3 *or* e2, since e2 initiates D which *is* preemptible by e3.

Algorithm 2.2 can still be used to determine the nominal preemption structure for the control structure's set of basic cs's. However, the output of Algorithm 2.2 must then be modified by removing preemptibility relations as specified.

2.8: Stopping the Flow of Control

Although the emphasis has been on how LC moves within a control structure, there may well be times when there is simply no work to be done for the moment. It is worth pointing out how the existing notation indicates this with some examples.

Basically LC will halt when it either:

1. Reaches the "end" of a control structure, and finds no '*', or
2. Reaches a slash ('/') beyond which no events (which are capable of interrupting the control structure to the left of the slash) have occurred.

Several examples are given in Figure 2.14 to clarify this concept; for conciseness, a typical (but not unique) task string which may be generated by each control structure is given. Additional notation should be self-explanatory.

((A*/e1)B)* ---> A A A e1 B A A A e1 B ...

((A/e1)B)* ---> A (wait) e1 B A (wait) e1 B ...

((A*/e1)B) ---> A A A e1 B (halt)

(((A*/e1)B)/e2)* ---> A A A e1 B (wait) e2 A A A ...

Fig. 2.14. Examples of processor idling.

2.8.1: Breaks in Event Coupled Lists

In light of the interpretation given to constructs which result in stopping the flow of control, it will be noted that there is no way to apply iteration to a portion of the control structure which includes all of a lower priority control structure and part of an event coupled list. What is needed is the concept of a *break*, which is essentially a restricted "go to" statement; it directs LC to jump over the rest of the event coupled list to the right parenthesis matching the initial left parenthesis of the event coupled list. Thus it enables the iteration at the end of the event coupled list to be applied to any intermediate part of the list as needed. The syntax for a break is the up-arrow (↑) at the point where the break is desired; it always follows a basic control structure, so it can be incorporated into that BNF:

<basic cs> ::= <task> | <basic cs> ␢ <task> | <basic cs> ↑

As an example, consider the control structure of Example 2.2 modified to include two breaks:

(A/(e1:((B↑/e2)C)|e3:((D↑/e4)E)))*

Now, when LC reaches the end of B or D, it returns to A instead of waiting for e2 or e4, respectively.

2.9: External Termination of a Control Structure

Consider the control structure:

Example 2.3. $((((A^*/e1)B^*)/e2)C)^*$

Since B^* is non-terminating and runs at a higher priority than A^*, A will never be executed again once e1 occurs.[1] There is nothing wrong with this per se, but with the given notation it is not possible to represent the case where occurrence of e2 *aborts* the repetition of B, and returns control to A^* after executing C rather than to B^*.

To do this, the notation must be able to indicate that occurrence of an event terminates execution of a particular control structure, and thus LC does not return to that control structure until its initiating event occurs again. The modified syntax:

```
<task> ::= <task id> | <non-preemptible tid> | <abort tid>
<abort tid> ::= @<task> | @(<ev list>)<task>
<abort cs> ::= @<closed cs> | @(<ev list>)<closed cs>
<closed cs> ::= ...(same as before plus: )... | <abort cs>
```

Thus it can be specified that *any* event aborts a task (e.g. @B) or set of tasks

[1] Recall that an event "flag," in this case e1, is not turned off until the *end* of the control structure which its occurrence initiates. B^* has no end.

(e.g. @(A B C)) or that any set of events causes termination (e.g. @(e2)B). The event which aborts the task(s) need not be the same as the one which causes preemption in a particular case; execution is terminated as long as the aborting event occurs sometime after preemption and before LC returns to the task.

If the control structure of Example 2.3 is changed to make B an <abort tid>, the desired behavior is obtained:

((((A*/e1)@(e2)B*)/e2)C)*

Now the string 'A A A e1 B B B e2 C A A A ...' can be generated, where repetition of A and B is for an arbitrary number of times.

2.10: Return of Control to a Preempted Task

There are two distinct choices of what to do when LC returns to a task which was interrupted during its execution: either resume execution from where it left off, or start over again from the beginning of the task. These two strategies will be referred to as *resumption* and *restarting* respectively. Each strategy has its advantages and may be the best choice in different situations. A task which is interrupted often enough may *never* complete if it is always restarted from the beginning. On the other hand, in a process control situation the inputs to an interrupted task may have changed radically since it was preempted, and resuming the computation started with the old inputs may lead to anachronistic outputs which are not relevant to the current control situation. Therefore, it is desirable to incorporate means of representing both strategies in the notation. For complete generality, it must be capable of handling a situation where two different tasks in the same control structure may follow the two different strategies. Furthermore, it is necessary to remember the point of interruption in the case of resumption, so the processor will know where to resume execution.

When the problem of *restarting* a control structure is examined carefully, it is seen that there are really two sub-cases which are of interest. First it must be recognized that the actual unit which is restarted is the task. At the next higher level, a task appears in a control structure as part of a basic control structure. Thus the problem is really how to restart a <basic cs>. If there is only one task in the <basic cs>, the problem is easily solved--simply restart that task. If there is more than one task in the <basic cs>, then the entire <basic cs> could be restarted from the beginning of its first task, or it could be restarted from the beginning of the task which was partially finished when the preemption occurred. For example, consider the following control structure:

$$(((A\ B)^*/e1)C\ D)^*$$

If event e1 occurs, and C D executes, (A B)* must be restarted (or resumed). Here are the possibilities:

1. Resume from the point of interruption, in either A or B.
2. Restart from the beginning of A.
3. Restart at the beginning of A if LC was at A when e1 occurred; restart at the beginning of B if LC was at B when e1 occurred.

The first case will be the default case, and !s assumed for all basic control structures as they have been so far defined. The second case will be called *global* *restart;* the third case *local restart.* If a syntax is defined for the concept of global restart, it can be used to synthesize local restart as a special case. Thus a syntax will be given called "restart cs", and it will have semantics of "global restart", the second case above.

<restart cs> ::= > <basic cs>

To control the scope of the restart symbol, restart control structures are introduced into other control structures strictly through their appearance in closed control structures:

```
<closed cs> ::= ( <basic cs> ) | ( <preemptible cs> ) |
        ( <closed cs> <preemptible cs> ) | ( <closed cs> <basic cs> ) |
        ( <closed cs list> ) | ( <restart cs> )
```

Here is an example of a control structure containing restarts:

(((()>A B)(C D)((>E)(>F)))/e1)G)*

Execution of this control structure proceeds identically to that of the basic control structure (A B C D E F) until event e1 happens. This causes execution of G; after G completes:

1. If LC was at A or B when e1 happened, LC returns to the beginning of A (global restart of (>A B)).

2. If LC was at C or D when e1 happened, LC resumes from the point of interruption in either C or D.

3. If LC was at E or F when e1 happened, LC returns to the beginning of E or F respectively (note that local restart of (E F) is equivalent to ((>E)(>F))).

2.10.1: Conditional Restart of a Control Structure

There is another possibility which should be represented. In some instances, a task should be restarted if it was preempted by one event (or one of a set of events), but resumed if it was preempted by another. This is handled by explicitly listing the events which would cause restart of a task. Thus a restart cs without an event list is unconditionally restarted, while one with an event list is only restarted if an event in its event list occurred since it was last run.[1]

<restart cs> ::= > <basic cs> | > (<ev list>) <basic cs>

Example:

(((((>(e2)A)*/e1)(>B))*/e2)C)*

Here A is restarted if either

1. A is preempted by e2 or
2. A is preempted by e1, which starts B. B is then preempted by e2 before completion.

B is unconditionally restarted, and A is resumed if e2 does not occur between the time of A's preemption by e1 and the resumption of A.

[1] Note that this means that the restart causing event need not be the one which caused the task's preemption; there may have been a chain of preemptions which included the restart causing event, and this is deemed sufficient cause for restart.

2.11: Codestripping

A time-sliced allocation of processor time can be represented with the existing notation by letting the event variables stand for timer-generated interrupts. One additional form of preemption which will be explicitly represented here is codestripping, as outlined in Section 1.1.

In codestripping, calls to the operating system are inserted into a task by the compiler at calculated intervals, resulting in preemption of the task when they are executed. The syntax is as follows:

<codestripped cs> ::= <basic cs> / <integer>

<preemptible cs> ::= <control structure> / <event list> | <codestripped cs>

Thus codestripped control structures are introduced into other control structures under the same syntax as preemptible control structures. An example of a codestripped control structure:

((A B/5)C)*

The meaning here is that the basic control structure A B is executed 1/5 at a time, based on its total (estimated) execution time; it is then preempted and C is executed. When C finishes, LC returns to the point of preemption, and executes another 1/5 of the way through A B (whether this is actually in A or in B depends of course on their relative lengths). Thus C will be executed five times for every single execution of A B.

Notice that control structures such as (>A B/10) are syntactically illegal; the notion of globally restarting (or locally restarting, for that matter) A B is incompatible with the semantics of codestripping. Furthermore, codestripping of closed control structures could lead to highly ambiguous or meaningless structures and is disallowed. This prevents such structures as ((A B/5)/10) and (((A B*/e1)C)/5). Structures which execute until they either finish a codestrip or are interrupted by an event *are* allowed, as they should be, e.g. (((A B/5)/e1)C)* which executes C for every 1/5 of A B executed and whenever e1 happens.

3: Representational Power of the Notation

3.1: Introduction

This chapter presents a catalog of control structure types which the notation of the preceding chapter is capable of representing. It is not claimed that every conceivable type of representable control structure is included, but the list attempts to be comprehensive as to general forms. Some examples are also given of types of control structures which are not representable.

3.2: Control Flow Graphs

Control structures can be conveniently categorized by the topology of their *Control Flow Graphs*, or CFG's. A CFG is a directed graph; more precisely, it is a set of *nodes* and *directed arcs*, where a node represents a basic cs and an arc represents the movement of LC between two nodes. The nodes bear the names of the basic cs's which they represent.

Consider an arc A which originates at basic cs *o* and has as a destination basic cs *d*. If *o* occurs to the left of *d* in the control structure, then arc A is a *forward* arc; otherwise, it is a *backward* or *back* arc. Either type of arc may bear labels:

1. An arc which represents the uninterrupted flow of control due to termination of a basic cs is a forward arc, and is unlabelled. Note that this includes breaks as detailed in Section 2.8.1.

2. An arc which represents the flow of control due to preemption

by an event occurring is a forward arc (an *event arc*) and is labelled with the corresponding event variable.

3. An arc which represents the flow of control due to iteration is a back arc and is labelled with an "*".

It may seem that tasks rather than basic cs's should be at the nodes of CFG's, and in fact the algorithms used for determining real-time latencies must sometimes deal with control flow at the task level. However, this additional detail adds nothing to the breadth of representable control structure types, and in fact detracts from the readability of the CFG's.[1]

Figure 3.1 gives an example of the CFG for a simple control structure.

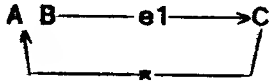

Fig. 3.1. CFG for ((A B)/e1)C)*.

A string naming the tasks and (optionally) the events encountered in a path taken by LC through a CFG is called an *execution* of the corresponding control structure. A B e1 C A B and A e1 C A e1 C are both executions of the above cs.

[1] If, for example, a basic cs is preemptible by event e*i*, then every task in the basic cs would have a forward arc labelled e*i*.

3.2.1: Priority Levels

As an extra benefit, the CFG notation provides a convenient mechanism for formalizing the concept of priority level, which has been used somewhat intuitively thus far. To find the priority level of basic cs *i*, do the following:

1. Let the leftmost basic cs in the control structure have priority *0* by definition.

2. Find the acyclic path from the priority *0* basic cs to basic cs *i* having the largest number of event arcs.

3. The priority of basic cs *i* is equal to the number of event arcs in this path.

3.3: Interrupt Driven Control Structures

The CFG's for control structures using only sequencing and iteration are fairly straightforward and do not expand the catalog of representable control structures by much. The sequence of tasks within a basic cs is implicitly represented, and forward control flow from one basic cs to another simply translates to an unlabelled arc in the CFG.

The more interesting CFG's are those which are derived from control structures having event variables. It is readily apparent that the notation has considerably more flexibility than that which is needed for representing traditional priority interrupt schemes. This flexibility is derived principally through the placement of the "*" iteration character and by use of the branching introduced by event coupled lists. The latter has been mentioned briefly; the former bears clarification.

A back arc can be originated from any basic cs by following it with an "*".

However, there is a degree of freedom in specifying the *destination* of the back arc; this will be exercised in enlarging the catalog of control structures. Fundamentally, the back arc may return to the same priority level, a lower one, or the lowest one. If it does not return to the lowest level, a certain "shrinkage" in the future range of LC is experienced. This will be elaborated on shortly. Additional variations on the fundamental types are achieved through use of the interrupt mask (non-preemptible tid), external abort and restart/resume capabilities.

3.3.1: Globally Cyclic Control Structures

Under this category is included all control structures with CFG's such that every back arc, regardless of its originating priority level, goes to the first task of the lowest priority level. Informally, this means that upon completion of the tasks at a given priority level, the processor will scan all the event variables in the control structure from the lowest level to the highest, and begin execution of the highest level task pending. This is as opposed to control structures with local cycles, where the lower priority events are not necessarily considered in each such situation.

The traditional interrupt systems available on most processors fall into this category; such systems are further subdivided into two types, which are called here the *weak* priority system and the *strong* priority system. In the weak priority system, although arbitration between interrupts from two or more events is provided, there is actually only a single true level of interruption. There is a "user" or "main" program which runs at the lower priority, and any number of events may each preempt it; however, no event may interrupt any task which gained control itself via an interrupt. This type of control structure is represented using event coupled lists, as in Example 3.1.

Example 3.1. (MAIN/(e1: A|e2: B|e3: C))*

The CFG (Figure 3.2) has an interrupt branch from "main" for every interrupting event, to the basic cs it initiates. Completion of A, B or C forces LC to return to MAIN, so there is a back arc from each of them. For the sake of keeping the CFG's readable, multiple back arcs with the same destinations will be combined, as is done in Figure 3.2. It is worth keeping in mind, however, that this does not imply that another type of node (junction) has been added.

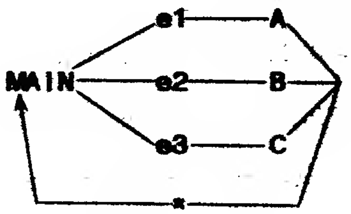

Fig. 3.2. CFG for Example 3.1.

A strong priority system supports a processor priority; the currently running task has a priority *n* associated with it, and any events interrupting with priority *m* > *n* may preempt it. With the exception of the ability provided for masking interrupts, the processor runs the highest priority task waiting for service at any time. This type of multiple priority level interrupt system is represented by strict nesting of preemptible (and iterative) control structures, as shown in Example 3.2.

Example 3.2. ((((A*/e1)B)*/e2)C)*

The general form can be recursively constructed; each "layer" looks like:

((<iterative cs>/<event var>)<basic cs>)*

which is itself an iterative cs. The <basic cs> runs at the next higher priority than the rightmost basic cs in the <preemptible cs>.

A CFG for Example 3.2 is given in Figure 3.3; it can be seen that the properties of nested interrupt systems have natural analogues in the graph:

1. Let α and β be basic cs's in the CFG. If there is an acyclic path from α to β whose last arc is labelled *ei*, then there is an arc from α to β labelled *ei*. This property stems from the transitivity of interruption in a nested, multiple priority system.

2. There is a back arc from the last basic cs at each priority level to the beginning of the lowest priority basic cs. After completion of the control structure at a given priority level, LC returns to the highest level with a pending request.

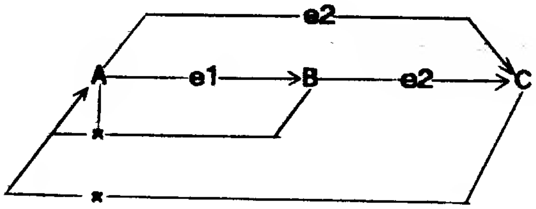

Fig. 3.3. CFG for Example 3.2.

3.3.2: Acyclic Control Structures

At the other end of the spectrum are found control structures with *no* back arcs; these represent completely non-iterative systems where the flow of control terminates when it reaches the end of any path. Such control structures are further subdivided into two types:

1. Linear control structures - control flow is straight-line and thus entirely predetermined, as in the example of Figure 3.4.

2. Branched control structures - real-time decisions based on event occurrences determine the actual flow of control; see Figure 3.5., which provides an example.

The subject of linear control structures does not leave much room for discussion and is included mainly for completeness. However, there are some interesting observations that can be made about branched control structures representable with the notation, and which apply independently of whether there are cycles present; these will be considered in the following section.

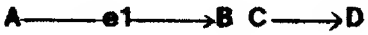

Fig. 3.4. CFG for the control structure ((A/e1)(B C)D).

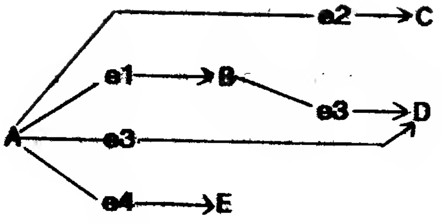

Fig. 3.5. CFG for the control structure (A/(e1:(B/(e2:C|e3:D))|e4:E)).

3.3.2.1: Branched Control Structures

It is interesting to note that while tree-shaped CFG's such as the one in Figure 3.6 can be represented, allowing arbitrary tree-shaped interrupt structures is not compatible with the transitivity of interruption. In fact, the notation cannot represent any tree of depth greater than one where the forward arcs are all event arcs. Thus a CFG such as the one in Figure 3.7 has no corresponding control structure.

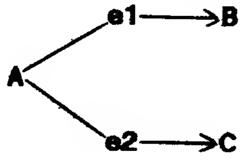

Fig. 3.6. A tree-shaped CFG, for (A/(e1:B|e2:C)).

For example, consider an attempt to derive a control structure for the CFG in Figure 3.7, a tree with a depth of 2. By Algorithm 2.2, it is found that since C interrupts B and B interrupts A, C must also interrupt A. Thus an arc labelled e2 must be added from A to C, and the tree structure is lost. Event e2 (and e3) can be masked from interrupting A; but then e1 is also masked, since it initiates B which is interruptible by e2. This same line of reasoning applies to any other attempt to produce a tree-shaped control structure of depth greater than 1.

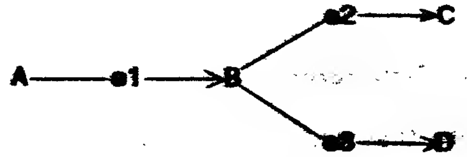

Fig. 3.7. A CFG which has no corresponding control structure.

Essentially, this restriction says that there cannot be control structures which have completely local preemption structures, and yet at the same time be initiated by some event. To incorporate this type of structure would require a notion of "local" and "global" events, with suitable restrictions on their scope. The additional complexity this would introduce may be incompatible with the attempt to keep the notation concise, but this may be a logical extension of the language for some applications.

Although it does not represent a preemption structure, Figure 3.8 shows a CFG which is similar to that of Figure 3.7, but which *is* representable, and by the following control structure:

('A B/(e1:C |e2: D))

The arc from A to B represents control flow on termination of A, but A cannot be interrupted.

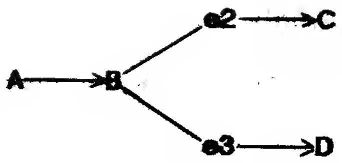

Fig. 3.8. A representable tree-shaped CFG.

3.3.3: Locally Cyclic Control Structures

Included in this class are all those control structures having back arcs which do not return LC to the lowest priority level task. This group is further subdivided into structures which *never* return control to the lowest priority task, and those which may or may not make the return at some point. While the emphasis here is on returning to the *lowest* priority level, the same sort of distinctions can be made about *any* priority level and its superiors. Examples of each case will be given.

3.3.3.1: Dynamically Decreasing the Range of LC

Consider the following general form of control structure:

Example 3.3. (... <preemptible cs><closed cs>*/<event var> ...)*

This has a non-terminating "<closed cs>*" construction, which corresponds to a back arc in the CFG from the end to the beginning of the closed cs. Although the rightmost "*" forces LC to return to the beginning of the control structure (if the "*" is reached), the <preemptible cs> will not be resumed since the following

<closed cs> runs at a higher priority, and is non-terminating.

Figure 3.9 gives the CFG for the control structure:

Example 3.4. $(((A/e1)(B\ C)^*/e2)D)^*$

which has the above general form. It can be seen that once a non-terminating loop is entered, although it may be preempted by higher priority tasks (either momentarily or permanently), control will not return to any task to its left. Thus the control structure has effectively "shrunk", in that certain tasks are no longer executable. This shrinking may occur in stages, if there are several events which initiate iterative control structures, and which occur in succession; or it may occur all at once, if the rightmost such event occurs first.

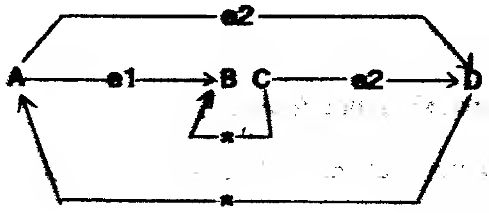

Fig. 3.9. CFG for Example 3.4.

3.3.3.2: External Termination of Local Cycles

A local cycle need not always indicate a decreasing control structure. If the "<abort cs>" construction is used, then control may reside for an arbitrarily long time in a given sub-structure (local cycle), and finally return to lower priority levels when the aborting event occurs. The control structure of Example 3.4 can be modified by the addition of a single "@" symbol:

Example 3.5. (((A/e1)@(B C)*/e2)D)*

Now when e2 occurs, it "shuts off" e1 as well as initiating D. This is a dynamic behavior and as such is not well suited to representation by a CFG; however the real-time latency algorithms must certainly take account of it.

3.3.3.3: Restrictions on Local Cycles

A back arc can be formed from the end to the beginning of any closed control structure, and hence there is little restriction on its range of possible destinations. One notable exception occurs in the presence of event coupled lists. Figure 3.10 gives a CFG which does not have a corresponding control structure; its illegality is the presence of a back arc which cuts across the "|" syntactic boundary in an event coupled list.

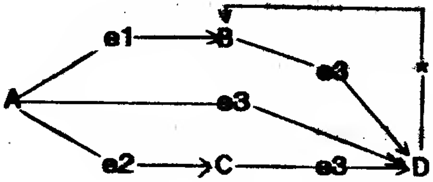

Fig. 3.10. CFG with an illegal back arc.

Essentially, this says that the forking caused by event coupled lists forms two or more independent sub-control structures, and LC cannot move freely from one to the other. However, it is possible that an event external to all the branches may preempt any of them; thus a CFG identical to that of Figure 3.10, except that it has no back arc, corresponds to the legal control structure:

(((A/(e1: B|e2: C))/e3)D)

3.4: CFGs at the Task Level

There are several variations on the general classifications presented here which arise principally when control flow at the inter-task level is considered. As previously mentioned, the complexity of the resulting CFG's limits their usefulness. Thus these variations are more suitably discussed in the context of latency algorithms; furthermore, they do not introduce new general classes of control structure types as far as the topology of their CFGs is concerned, but instead result in perturbations of those already considered.

However, it is reasonable to examine the changes which would be induced on a CFG which has single tasks at its nodes, rather than basic cs's. Use of the "<non-preemptible closed cs>" or "<non-preemptible tid>" constructions results in the removal of the appropriate event arcs. In addition, if the task immediately prior to the "/<event var>" construction is masked, an unlabelled forward arc is added to show the flow of control which occurs on termination of the masked task.

The default mode of control return to a preempted task is resumption, as discussed in Chapter 2. Thus any arc (backward or forward) to a preemptible cs of this type must be dynamically relocated to point to the task which was in execution when preemption occurred. Again, this is not easily representable with a static CFG, and in fact corresponds to the need to store some "state" information while a task is dormant.

If a task is to be restarted, this problem does not arise; in fact, if an entire closed cs is of restart type, there will be no arcs pointing to tasks internal to the closed cs which originate outside of it. The only entry point from the external world's point of view is the beginning of the initial task.

4: Real-time Properties of Control Structures

4.1: Introduction

A primary motivation behind developing the language presented in Chapter 2 is to provide a representation of control structures suitable for use as an analytical tool. Specifically, it provides a convenient format for conveying preemption and control flow information to an algorithm which then determines real-time properties of the given control structure.

The algorithms to be given here are not intended to provide an exhaustive analysis of a control structure, but rather to be representative of the types of analysis which may be performed. The real-time properties measured here are of common interest; however, it will probably be the case that, depending on the needs of the particular user, different real-time properties may be of special interest. In many cases, the given algorithms can be adapted for measuring different intervals with minimal changes. In other cases totally new algorithms may be needed, but parts of those given will still be useful.

Much of the terminology used here was developed in [Teixeira 78] and the reader is referred there for a complete discussion.

A principal goal here will be to develop algorithms for determining the *worst case latency* of a list of tasks in a given control structure. Informally, the worst case latency of a list of tasks α (written $l(\alpha)$) is the longest time that can elapse without there being a complete execution of each task in the list in the order given. The list of tasks whose latency is being measured will be referred to as a *constraint*. The latency of a constraint is measured with respect to an *execution* of a given control structure, where an execution is a list of tasks in the order in which they are executed by the CPU in a particular invocation of that control structure. Each element (task id) of the execution has a *weight* associated with it, written as |<task id>|. The weight represents an upper bound on that task's execution time on a particular processor.

Note that depending on event timings, a number of different executions (of finite or infinite length) may be generated by a single control structure. Consider the control structure:

$$(((A\ B)^*/e1)C)^* \qquad (5.1)$$

Possible executions include:

A B A B A B ...

A B C A B C ...

A B A B C A B A B C ...

among many others. Also note that in the case of preemption a task may be suspended and restarted, and thus partial weighting (or its effective equivalent) must be accounted for.

The weight of a list of tasks is the sum of their individual weights. The worst case latency of a constraint α with respect to an execution β, is the sublist of β with greatest weight which does not *contain* α. The term "contains" as used here means that the elements of α occur in order and with their full weights; there may be arbitrarily many other tasks interleaved. For example, (A B C D) contains (A C) as well as (A B), but it does not contain (C B).

The provision that the tasks be included with their full weights is emphasized for the following reason. In many real-time process control applications, the inputs to a task may change at any time, but the scheduling of task initiation may not be synchronized with the arrival of new inputs. Thus it is entirely possible that new inputs may arrive immediately after the initiation of a task, i.e., after it has already read the outdated inputs. Given this possibility, it may be that nearly *two* complete occurrences of the constraint may be executed in an interval which still does not contain (in the strict sense defined above) a single occurrence of the list. For example, given the control structure (A B C)*, consider the execution A B C A B C. If an input to A arrives immediately after A reads its old input,[1] then it is only after the *second* occurrence of C has completed its execution that all the tasks in the constraint will have been executed in order (the constraint is *satisfied* by such an execution). Thus a way is needed to represent an execution whose end-tasks are weighted just less than their nominal values; the notation chosen is bracketing such a task on its "short side"; [A means "begin just after the start of A", and C] means "stop just before the finish of C". The weight of such a task is its nominal weight minus ϵ, where ϵ is arbitrarily small. Thus the worst case latency of (A C) in (A B C)* is | [A B C A B C] |.

The list (A B C A B C) is an example of a *critical window* for (A C), where a

[1] Unless it is known that the timings of such data arrivals can be synchronized with task initiation, it must be assumed that this could occur at *any* time after A is initiated.

critical window is defined as a list α such that α contains two occurrences of a constraint C but $[\alpha]$ contains no occurrences of C. In many cases the worst case latency of a constraint will turn out to be the weight of a critical window (the *most critical window*). The worst case latency of a constraint with respect to a control structure (as opposed to an execution) is taken over all the possible executions that may be generated by the control structure – no matter what the event timings (within specified limits), there can be no longer interval which does not contain the list. Thus part of the problem faced is to classify the types of executions which may be generated by a control structure and narrow the choice among them for finding the worst case, since otherwise the combinational explosion in the number of possible executions would make the problem intractable.

4.2: Weights of Task Identifiers

It was mentioned briefly above that a weight is associated with every task identifier, representing an upper bound on its execution time. Naturally this must be with respect to a particular processor, but even with this restriction there are some difficulties in determining a meaningful upper bound on execution time. Aside from input dependent computation times, there are processor dependent variables such as memory access time in a virtual storage system. The worst case time would occur when all memory references were to the slowest storage device, but the probability of such a case actually occurring may be nearly zero. On the other hand, there may be an uncomfortably large variance associated with the mean access time when critically time-dependent processes are involved. It seems then that in such a case one must either arrive at a statistically reasonable upper bound on memory access time or change the storage allocation parameters of time dependent tasks to ensure their residence at a particular level or above (in access speed) of the storage hierarchy.

If an upper bound on the execution time for a task does not exist, this would imply potentially infinite worst case latencies and there would be no purpose to applying the algorithms given here. If there is any question of the value of an upper bound, then it must be chosen carefully in light of the particular application of the latency information. The weight of each task will be an input to the latency algorithms along with the control structure, and it will be assumed that a function (table look-up) exists which returns this weight in response to the notation |<task id>|.

4.3: Properties of Event Variables

In order to arrive at worst case latency times for a control structure containing event variables it is necessary to know something more about the timing of the events represented. To illustrate, consider the control structure:

$$(((A\ B)^*/e1)C\ D)^* \qquad (5.2)$$

If e1 never occurs, the only possible execution of this control structure is (A B A B A B ...). The latency $l(A\ B)$ in this case is $2(|A|+|B|) - \epsilon$, since the longest sublist which does not contain A B would be [A B A B]. On the other hand, if e1 occurs at least once every $|C|+|D|$ seconds, then $l(A\ B)$ is infinite, since the only execution generated is (C D C D C D ...) (ignoring possible initial executions of A and B). If the control structure contains more event variables it may become difficult to determine the worst case latency (the largest l(A B)) by inspection, and the need for additional information about the event variables is clear.

In particular, what is needed is the following:

1. $\tau_{min}(e_i)$: the minimum period of event e_i; it is guaranteed that e_i will not occur more than once in any interval of $\tau_{min}(e_i)$ seconds.

2. $\tau_{max}(e_i)$: the maximum period of event e_i; it is guaranteed that there will be at least one occurrence of e_i in any interval of $\tau_{max}(e_i)$ seconds.

It is entirely plausible and indeed likely that in some situations $\tau_{min}(e_i)$ will be the same as $\tau_{max}(e_i)$. This is the case for all regularly occurring cyclic events, such as data sampling, processor time slicing, etc.

In general, it is impossible to distinguish a $\tau_{min}(e_i)$ which is less than the processor instruction cycle time from an infinitesimal one since the processor could not possibly respond to an event which occurred at that rate in any meaningful way. In fact, for a reasonable system, one would have to pick a τ_{min} (e_i) considerably larger than the instruction cycle time, but the actual value will be application dependent. For most events of interest it will be possible to determine a reasonably tight $\tau_{min}(e_i)$; e.g., if the event represents an I/O service request, it cannot occur more often than some time interval dependent on the I/O device's maximum character transmission rate.

Unfortunately, finding a good value for $\tau_{max}(e_i)$ is more difficult in many cases.

An event often represents an exceptional condition, which may never arise in particular executions. Fortunately, most control structures will not put time critical tasks in such a position that their initiation depends on $\pi_{max}(e_i)$, but rather it is more likely that the completion of a constraint may be influenced by time lost after such an event occurs; and the time lost will be a function of π_{min} (e_i), not $\pi_{max}(e_i)$. If a good value of $\pi_{max}(e_i)$ is not available for a particular event, then it is more likely that the interval of interest would be the maximum time from the occurrence of e_i to the initiation and/or completion of its associated control structure, rather than the longest time between such executions (a latency value).

5: Algorithms

5.1: Introduction

A series of hierarchically related algorithms will be presented in this chapter, which will be directed at the problem of finding the worst case latency of a constraint with respect to a given control structure. Each algorithm in the hierarchy is applicable to a larger subset of the set of all representable control structures, and may call upon the algorithms designed for solution of the problem on a lesser subset as subroutines.

The overhead due to context switching is not explicitly taken into consideration here. It may be accounted for by a fractional reduction of the effective processing power of the CPU, when computing the worst case task weights. If this is not satisfactory, then the algorithms could be adjusted so that each event occurrence and corresponding initiation is counted, and the overhead due to each could be added to the delays attributed to interruption.

As the worst case latency algorithms are developed, it will be seen that the determination of algorithms to measure several other real-time properties, interesting in their own right, is required. Finally, special cases may result in substantial simplification to the algorithms, and examples of this effect are included.

5.2: Latencies in the Absence of Preemption

The first step taken here toward the general solution of the worst case latency problem is the development of algorithms to determine the latencies when no preemption is present, i.e. when there are no event variables or codestrips in the control structure. This leaves control structures which generate finite and infinite lists of tasks, in which all tasks execute to completion once initiated.

Since only non-terminating iteration is represented (in the absence of preemption), all finite lists must contain no iterative components. Furthermore, any finite list L of tasks which contains at least one occurrence of a constraint C[1] may be broken down into a series of possibly overlapping sublists:

$$(\beta_1, \alpha_1, \alpha_2, \cdots, \alpha_n, \beta_2) \tag{5.1}$$

with respect to a constraint C where:

1. β_1 and β_2 each contain one instance of C, but $\beta_1]$ and $[\beta_2$ contain no instances of C.

2. The α_i's are critical windows for C.

The sublist β_1 is the head of the list L having minimum weight and which also contains one instance of C; β_2 is the tail with least weight which contains one instance of C. The list α_1 is the critical window which starts at the first instance in L of the first task in C; α_i is the critical window which starts at the *ith* instance in L of the first task in C. If L contains no critical window, there will be no α_i's;

[1] If L does not contain C, then the latency of C in L is infinite.

similarly, if L begins or ends with a critical window then β_1 or β_2 respectively may also be empty.

Figure 5.1 gives an example of the breakdown for the list (A B C D B C B C E) and the constraint (B C). Note the overlapping of the sublists, and that in this case $|\alpha_1| \neq |\alpha_2|$.

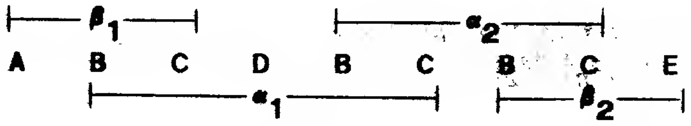

Fig. 5.1. Breakdown of a finite task list into sublists.

Theorem 5.1. The latency of a constraint C with respect to a finite list L which contains at least one occurrence of C is the maximum weighted sublist in the set of sublists $\{\beta_1, \alpha_1, \cdots, \alpha_n, \beta_2\}$, where the α_i's and β_i's are as defined above.

Proof. The proof will be given in two parts; first, by showing that any list which contains at least one occurrence of C can be broken down in the above manner to obtain such a set of sublists which includes all the tasks in the original list; and second, by showing that no other sublist not in the set can have a greater latency for C.

The proof of the first part is given by showing a method of constructing the set $\{\beta_1, \alpha_1, \alpha_2, \cdots, \alpha_n, \beta_2\}$ given such a list L and a constraint C.

Find each sublist of L which exactly contains one instance of C (i.e., a sublist γ such that γ contains C but $[\gamma$ and $\gamma]$ do not contain C); label each such sublist γ_i, for $i = 1$ to n, where n is the number of instances of C in L. The list L can thereby be considered as a series of sublists;

$$\phi_1, \gamma_1, \phi_2, \gamma_2, \cdots, \phi_{n-1}, \gamma_{n-1}, \phi_n \qquad (5.2)$$

where the ϕ_i's do *not* contain C and may be empty. If γ_i overlaps γ_{i+1}, then ϕ_{i+1} will be empty. This set of sublists includes every task in L, and with no permutation of the original ordering. Then:

1. β_1 is ϕ_1 appended to γ_1.

2. α_i is the list starting at γ_i and continuing to the end of γ_{i+1}, including ϕ_{i+1}. Note that since γ_i and γ_{i+1} may overlap, α_i is *not* their concatenation.

3. β_2 is γ_{n-1} appended to ϕ_n.

Now for the proof that the worst case latency of C in L is the maximum of $\{|\beta_1|, |\alpha_1|, |\alpha_2|, \cdots, |\alpha_n|, |\beta_2|\}$.

Since the α_i's are all the critical windows in L, they represent all the lists α such that $[\alpha]$ cannot be expanded in either direction without the resulting interval containing C. Similarly, $\beta_1]$ and $[\beta_2$ cannot be expanded on their bracketed sides without introducing C to the interval. Since the concatenation of $(\beta_1, \alpha_1, \alpha_2, \cdots, \alpha_n, \beta_2)$ contains L, and none of these sublists can be expanded without the resulting sublist containing C, the only possibility for the existence of a sublist with greater latency is that there is such a list which includes parts of two of the above sublists. That such a sublist with greater latency does not exist is demonstrated by case analysis.

Again, consider the list L reorganized as in equation (5.2). Now, suppose that there exists a sublist ψ which includes part of β_1 and part of α_1, and that $|\psi| > |\beta_1|$ and $|\psi| > |\alpha_1|$. The sublist ψ cannot begin in ϕ_1, or it could not contain anything past γ_1 (without containing C) and hence $|\psi| < |\beta_1|$. But if ψ starts at the beginning of γ_1, it could include no more than α_1, and therefore $|\psi| \leq |\alpha_1|$. If ψ begins past the beginning of γ_1, it cannot contain anything past γ_2, and hence $|\psi| < |\alpha_1|$. Thus such a sublist ψ does not exist.

The same line of reasoning will show that a sublist with greater weight than any of $\{\beta_1, \alpha_1, \cdots, \alpha_n, \beta_2\}$ cannot being constructed from parts of adjacent α_i's, or α_n and β_2. Thus the worst case latency of C in L will be the maximum of $(|\beta_1|, |\alpha_1|, \cdots, |\alpha_n|, |\beta_2|)$. □

Algorithm 5.1, FLATENCY, summarizes the procedure to be followed in finding the worst case latency of a constraint C with respect to a finite list L.

Algorithm 5.1. FLATENCY(L, C)

Inputs: L, a list of task identifiers (a basic control structure); L[*i*] is the *i*th task in L.

C, the constraint (also a list of task identifiers); C[*i*] is the *i*th task in C.

Outputs: (l(C), start_index, finish_index);

l(C) is the worst case latency of C in L.

start_index is the index of the first task of the sublist of L which displays the worst latency for C.

finish_index is the index of the last task of the sublist of L which displays the worst latency for C.

Method:

1. Scan L to find:

β_1, the head of L with least weight which contains C.

α_i, $i = 1$ to n where n is the number of occurrences of C in L minus 1.

β_2, the tail of L with least weight which contains C.

This is accomplished as follows. All scans start from the *mark point*, initially L[1].

a. Reset the mark point to be the first occurrence of C[1] found during each scan. If no occurrence of C[1] is found, the mark point is set to the task past the end of the current scan.

b. β_1 is found by scanning until a complete occurrence of C has been found.

c. The α_i's are the lists which exactly contain two occurrences of C; they are found by scanning from the mark point for one occurrence of C, and then scanning from the new mark point for the second occurrence of C.

d. β_2 is the result of the final scan if no tail of L is a critical window.

e. If no occurrence of C is found in L, return (∞, -1, -1).

2. The weights of each sublist are accumulated during each scan, as well as the start_index and finish_index for that scan. At the end of each scan the weight is compared to the largest found so far, and saved as the new maximum $l(C)$ if it is greater (in which case start_index and finish_index are updated to the values for the just scanned list).

3. Return the final values (MAXIMUM($|\beta_1|$, $|\alpha_1|$, $\cdots$, $|\alpha_n|$, $|\beta_2|$), start_index, finish_index).

5.3: Latencies of Constraints in Cyclic Control Structures

In the specified language an infinite list of tasks is generated by the iteration construct; iteration is either applied to an entire control structure or to the last closed control structure in a <closed cs list>. Thus infinite lists are either entirely cyclic (the entire structure is repeated):

$$(A\ B\ C\ D\ E)^* \tag{5.3}$$

or have a start-up period followed by a steady state cycling:

$$(A\ B\ C)(D\ E)^* \tag{5.4}$$

It would be indeed unfortunate if the entire infinite list had to be examined to find the worst case latency, but due to the restrictions on its cyclic nature only a reasonably small number of cycles (to be determined) have to be examined to find the worst case. Thus the intention here is to reduce the case of an infinite list to a finite list which contains the worst case, and use Algorithm 5.1, FLATENCY, on the result.

The principle question is thus to determine how many cycles of the iterative portion of the list need be appended to the non-iterative portion (if there is one) in order to generate a list containing the worst case latency of a specified constraint. First, though, it must be determined whether or not the latency is infinite (assuming no task id has infinite weight).

Lemma 5.1. Given a control structure $(\phi)(\psi)^*$ and a constraint C, the worst case latency of C in $(\phi)(\psi)^*$ is infinite iff C contains a task A which is not contained in ψ.

Proof. If ψ does not contain a task A which is in C, then $(\psi)^*$ is an infinitely long list (and hence of infinite weight) which does not contain C, and thus in which C has infinite latency.

If ψ does contain every task in C, then if C contains n tasks at least every n repetitions of ψ contains C and hence the latency of C in $(\phi)(\psi)^*$ could not be infinite. □

Once it has been established that the latency is not infinite, the following theorem can be applied to find the sublist which contains the sublist with the worst case latency.

Theorem 5.2. Given an iterative control structure L = $(\phi)(\psi)^*$ and a constraint C containing n task identifiers, then if the latency of C in L is not infinite, the list formed by appending n + 1 copies of ψ to ϕ contains the sublist with the worst case latency for C in L.

Proof. Theorem 5.1 established that the worst case latency of a constraint in a list of tasks was either a critical window α_i or a head or tail of the list β_1 or β_2. By Lemma 5.1, if the latency is not infinite then ψ contains every task in C. Therefore

$$\beta_1 \subseteq \phi, \psi^n \qquad (5.5)$$

where ψ^n means n copies of ψ appended to each other. This is true since n copies of ψ must contain C, since each ψ contains each task identifier in C. Note that β_1 might be wholly contained in ϕ, nonetheless.

By similar reasoning:

$$\phi, \psi^{n+1} \tag{5.6}$$

contains the most critical window of $(\phi)(\psi)^*$; if the most critical window is contained in ϕ, then equation (5.6) must contain it. Otherwise, it is contained in $(\phi)(\psi)^*$. If the most critical window starts in ϕ but ends in $(\psi)^*$, then it cannot go any further than ψ^n since the first n copies of ψ must contain C; thus equation (5.6) contains the most critical window if this is the case also.

Finally, suppose that $(\psi)^*$ contains the most critical window. Consider the list θ formed by starting at the first occurrence of C[1] in the first copy of ψ, and ending at the last occurrence of C[n] in the n + 1st copy of ψ. The list θ must contain two occurrences of C, since ψ_1 through ψ_n contain C, and ψ_2 through ψ_{n+1} contain C. If [θ] contains no occurrences of C, then θ is a critical window. If θ is a critical window, then no critical window can exist which is larger than θ since it would have to be constructed out of more than $n + 1$ copies of ψ and thus would contain θ. Thus if θ is a critical window, it is the most critical window in $(\psi)^*$. But if θ is not a critical window, then it must contain a critical window, and by the same logic this critical window must be the most critical window in $(\psi)^*$. □

Algorithm 5.2, ILATENCY, shows how to use Theorem 5.2 coupled with the algorithm FLATENCY to determine the worst case latency of a constraint with respect to any control structure which does not contain preemption.

Algorithm 5.2. ILATENCY(L, C)

Inputs: L, a control structure which does not contain preemption.

C, a constraint (list of task identifiers).

Outputs: (l(C), start_index, num_tasks)

l(C), the worst case latency of C in L.

start_index, the index in L of the first task of the list whose weight is l(C).

num_tasks, the number of tasks in the list whose weight is l(C).

Method:

1. If L is not iterative, let (l(C), start_index, finish_index) = FLATENCY(L, C); return(l(C), start_index, finish_index - start_index + 1).

2. If L is iterative, then divide L into its iterative and non-iterative (if any) parts: $L = (\phi)(\psi)^*$.

   a. If ψ does not contain every task in C (not necessarily in order), return(∞, -1, -1).

   b. Let $K = \phi, \psi^{n+1}$ where n is the number of tasks in C. Let (l(C), start_index, finish_index) = FLATENCY(K, C); return(l(C), start_index, finish_index - start_index + 1).

5.4: Latencies of Constraints in Preemptible Control Structures

The next complication to be dealt with is the presence of event variables and multiple priority levels, implying the possibility of preemption before completion of a constraint, and thus additional weight for the worst case latency. In fact, at this point the possibility of infinite latencies arises due to lockout by higher priority tasks, even though the constraint may be contained in an iterative portion of the control structure.

The general case of preemptible control structures contains many additional complexities, if one includes external termination of control structures, non-preemptible tasks, codestripping, restarting, and idle time due to stopping the flow of control. Thus, in keeping with the theme of building a hierarchy of algorithms which handle increasing complexity with each new layer, the applicability of the next algorithm is restricted to include all the control structures allowable as inputs to ILATENCY, plus those containing <event list>'s (<event var>'s and <event coupled list>'s). Specifically there are the following restrictions:

1. No external termination (<abort tid> or <abort cs>).
2. No restarting of control structures (<restart cs>).

3. No codestripping (<codestripped cs>).

4. No non-preemptible tasks (<non-preemptible tid> or <non-preemptible closed cs>).

5. No stopping of LC. The highest priority ready task must always be initiated without delay. Thus a control structure such as:

$$((((A^*/e1)B)/e2)C)^* \qquad (5.7)$$

is illegal but

$$((((A^*/e1)B)^*/e2)C)^* \qquad (5.8)$$

is not. Event coupled lists must contain breaks (cf. Section 2.8.1) to ensure that waiting for higher priority events in the event coupled list does not occur.

6. Constraints must be contained wholly in a *subcontrol structure*, defined as a series of basic cs's, an iterative cs, or closed cs lists at a single priority level. In CFG terms, a subcontrol structure is an acyclic path through the control structure's CFG which contains no event arcs, back arcs or breaks. This allows all processor time spent at any other level to be treated as an addition to worst case latency, and lets the details of exactly which tasks are contributing to the increase be ignored. Additionally, the tasks of the constraint must not be contained in more than one subcontrol structure. If they are, then the worst case latency in the entire control structure would be ≤ the *minimum* of the worst case latencies in each subcontrol structure which contains the constraint; thus the present algorithms still give an upper bound. The problem here is that if the constraint can be satisfied by an execution which spans two or more priority levels, then the tasks being executed during preemption must be identified, and can no longer be lumped together and treated as time lost to interrupts.

7. Infinite event queues. An infinite number (or some suitably high number representing the maximum possible number of pending events) of occurrences of each event are remembered. This means that if an event happens before the previous occurrence has been cleared (by completion of the initiated control structure), the new occurrence will be held in a queue and not ignored.

5.4.1: Definitions and General Approach

The addition of preemption to a control structure introduces several interesting timing questions. For example:

> 1. The worst case latency of a constraint as previously defined, i.e. the longest time that can pass without their being a complete execution of each task in the constraint in order. This may now be prolonged by *initiation delay* as well as preemption delay. Initiation delay is time lost due to the initiating event not yet having occurred.
>
> 2. The worst case latency of an event, defined as the longest time that can elapse between the occurrence of an event and the start of the subcontrol structure which it initiates. What exactly constitutes the initiation of a subcontrol structure will be implementation dependent.
>
> 3. Related to (2), it may be desired to know the worst case execution time of a list of tasks at a given priority level; this is their execution time in the absence of preemption plus the most possible time lost to preemption. This may be more useful than (1) in cases where occurrence of an event signals the arrival of new data, rather than assuming that task initiation is unsynchronized with data arrival times.

In all these cases it will be necessary to make some assumptions which could lead to an upper bound which is somewhat greater than the actual worst case (in addition to the uncertainty in the estimate of worst case task execution time). In particular, Teixeira has shown [Teixeira 78] that the worst case occurs when all interrupting events happen at the beginning of an interval and continue happening at their maximum rate. It may be that the phase relationships of the events coupled with the execution times of their subcontrol structures is such that the events could never all happen together; if this is known in a particular case then its worst case may be different, and the initial phases of the events could be adjusted accordingly. The algorithms do allow specification of event phases, as will be seen. In any case the algorithms do give an upper bound to the problem.

The worst case latency of a constraint which executes at priority 0 (the lowest priority) can be determined in terms of nominal time in the absence of preemption plus time lost to interrupts; the initiation delay need not be considered. The fundamental difference between tasks at priority 0 and priorities greater than 0 is that if the worst case latency of a constraint involves more than one execution of tasks at a priority level greater than 0, there may be delay due to initiation of that priority level (which must be figured according to π_{max} of the initiating event in the worst case) for the additional task executions. The lowest priority level is assumed to be always running or ready, and thus has no such delay.

In general there will be some thought required to pinpoint the worst case for any time interval of interest; once determined, the algorithm to measure such a time interval can be constructed using the following basic technique.

> 1. Determine the relative priorities of every basic cs in the overall control structure, and associate with each event variable the subcontrol structure which it initiates (cf. Section 2.6.5). The priority of a subcontrol structure and its initiating event are the same. It is assumed that π_{min} and π_{max} are known for each event (cf. Section 4.3).
>
> 2. Determine whether the time interval (latency or otherwise) is infinite. This may be done in two steps:
>
> > a. If the time interval is infinite in the absence of preemption (determined as previously shown), then it is infinite in the presence of preemption.
> >
> > b. Otherwise, find out whether higher priority tasks can sufficiently load down the processor so that the interval of interest is never completed. One method for doing this will be shown.
>
> 3. If it is not infinite, determine the interval in the absence of

preemption and other delays.

4. Factor in the loss of time due to preemption and other delays; lifting any of the restrictions given in Section 5.4 will usually be seen as perturbations of this factor.

5.4.2: Finding Infinite Latencies

The control structures represented here provide no a *priori* method of guaranteeing fairness if preemption is present; i.e., it is entirely possible that in the worst case some tasks in the control structure may never be executed due to preemption by higher priority tasks.

Fortunately it is possible to determine whether this is the case in advance and at low computational cost, and this must be done before continuing with the analysis. If the latency at a given priority level is infinite then the iterative solutions to be used for solving for loss of time due to preemption do not converge. The method used is to determine a load factor for each subcontrol structure that can preempt a given one, and if the load is ≥ 1 then the given control structure's tasks will never execute.

In order to find the load factor due to a subcontrol structure $\not\downarrow$ with initiating event $e_{\not\downarrow}$, it is necessary to partition the set of events in the overall control structure as follows:

1. E_{always}; the set of events which can always preempt $\not\downarrow$, but can never be preempted by $e_{\not\downarrow}$. These are the events of higher absolute priority than $e_{\not\downarrow}$, as found by Algorithm 2.2.

2. E_{win_tie}; This is the set of events which cannot preempt $\not\downarrow$

and cannot be preempted by $e_{\downarrow}$, but are chosen over $e_{\downarrow}$ if $e_{\downarrow}$ and one of them are both pending at the same time. This set is the union of the following sets:

a. Events which have the same absolute priority as $e_{\downarrow}$, but occur to its left in the same event coupled list.

b. Events which have the same absolute priority as $e_{\downarrow}$, but occur in a different event coupled list which is entirely to the left of the event coupled list containing $\downarrow$.

c. Events which have higher absolute priority than $e_{\downarrow}$ but occur in an event coupled list which does not contain $e_{\downarrow}$.

3. E_{lose_tie}: This is the set of events which cannot preempt $\downarrow$ and cannot be preempted by $e_{\downarrow}$, but $e_{\downarrow}$ is chosen over one of them if both are pending at the same time. This set of events is the union of the following sets:

a. Events which have the same absolute priority as $e_{\downarrow}$, but occur to its right in the same event coupled list.

b. Events which have the same absolute priority as $e_{\downarrow}$, but are in a different event coupled list which is entirely to the right of the event coupled list containing $\downarrow$.

c. Events which have a lower absolute priority than $e_{\downarrow}$, but occur in an event coupled list which does not contain $e_{\downarrow}$.

4. E_{never}: This is the set of events which can never preempt $e_{\downarrow}$, and initiate subcontrol structures which can always be preempted by $e_{\downarrow}$. These are the events of lower absolute priority than $e_{\downarrow}$.

As an example, consider the control structure:

$$(A/(e1:B/(e2:C|e3:D)|e4:E/(e5:F|e6:G)))^* \qquad (5.9)$$

Its preemption structure appears in Figure 5.2, and the partitioning of its events in Figure 5.3.

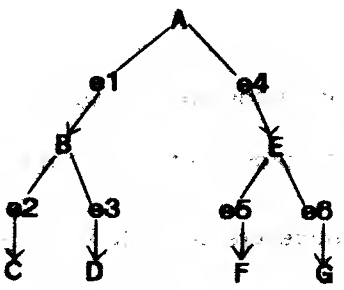

Fig. 5.2. Preemption structure for (5.9).

Initiating Event /Task	E_{always}	E_{win_tie}	E_{lose_tie}	E_{never}
none/A	e1, e2, e3, e4, e5, e6	none	none	none
e1/B	e2, e3	e5, e6	e4	none
e2/C	none	none	e3, e4, e5, e6	e1
e3/D	none	e2	e4, e5, e6	e1
e4/E	e5, e6	e1, e2, e3	none	none
e5/F	none	e2, e3	e1, e6	e4
e6/G	none	e2, e3, e5	e1	e4

Fig. 5.3. Partitioning the events of (5.9).

To decide whether a task A at a given priority level in a control structure may never execute, partition the events of the control structure relative to A as just described. Each event initiates a subcontrol structure (at a single priority level); let e_i initiate subcontrol structure ψ_i. The worst case load of a given subcontrol structure on the processor occurs when its initiating event happens at its maximum frequency:

$$\text{Worst case load}(\psi_i) = \frac{|\psi_i|}{\tau_{min}}(e_i) \qquad (5.10)$$

The total load factor is the sum of the worst case load factor for each event which might participate in the blockout of A; this is the set $E_{preempts}$ = $\{E_{always} \cup E_{win\_tie}\}$, since these are exactly those events which consistently get control over e_A no matter how long e_A may have been waiting in queue. Of course, if A is in the lowest priority control structure, there is no e_A and the set E_{win_tie} is empty; but the analysis of possible blockout due to preemption is unchanged. Let the events in $E_{preempts}$ be $\{e_i, \cdots, e_j\}$; then the total load factor is:

$$\text{Total load factor(A)} = \sum_{k=i}^{k=j} \frac{|\psi_i|}{\tau_{min}(e_i)} \qquad (5.11)$$

If the total load factor is ≥ 1.0, then the task A (and any other task in the same basic cs as A) never gets executed; its worst case latency is infinite. All the following algorithms assume that this check has been made before they are called, so that a finite solution is known to exist.

5.4.3: Delay Due to Preemption

The problem of determining the time taken up by preemption lends itself naturally to an iterative solution. In the worst case it must be assumed that every interrupting event happens at its maximum frequency (once every τ_{min} seconds). As the tasks initiated by one interruption are being executed, there may be additional event occurrences, causing further delay, etc. By equation (5.11), if the load factor is < 1 it is guaranteed that at some point the task in question (the one being preempted) will execute; but it is not clear when and for how long before it is preempted again.

The problem is then to solve for the total time taken to execute some set of tasks ψ of total weight W_ψ, in the presence of a set of interrupting events $\{e_i, \cdots, e_j\}$ which all happen at time zero and then again every $\tau_{min}(e_i)$ seconds, each initiating subcontrol structures with weights $\{W_i, \cdots, W_j\}$. The total time, T_ψ, is:

$$T_\psi = W_\psi + \sum_{k=i}^{k=j} \left(\left\lceil \frac{T_\psi}{\tau_{min}(e_k)} \right\rceil W_k \right) \tag{5.12}$$

The ceiling function is chosen since the quotient

$$\left\lfloor \frac{T_\psi}{\tau_{min}(e_k)} \right\rfloor \tag{5.13}$$

gives the number of occurrences for e_k in the interval $[T_\psi$; but since all events happen at the beginning of the interval (in the worst case) one additional occurrence must be added

$$1 + \left\lfloor \frac{T_\psi}{\tau_{min}(e_k)} \right\rfloor \tag{5.14}$$

but if the event occurs at the exact end of the interval T_ψ this occurrence must not be counted since γ will already be completed -- thus the choice of

$$\left\lceil \frac{T_\psi}{\tau_{min}(e_k)} \right\rceil \tag{5.15}$$

A quick iterative solution to (5.12) is had by noticing that an excellent lower bound is the solution to

$$T_\psi \geq W_\psi + \sum_{k=i}^{k=j} \frac{T_\psi W_k}{\tau_{min}(e_k)} \tag{5.16}$$

which is

$$T_\psi \geq \frac{W_\psi}{1 - \sum_{k=i}^{k=j} \frac{W_k}{\tau_{min}(e_k)}} \tag{5.17}$$

Notice that the denominator is exactly 1 - Equation (5.11), the total load factor, which has already been computed. Equation 5.17 implies that running γ with interrupts is like running ψ on a processor whose strength has been diminished by the load factor of the interrupting tasks.

Thus equation (5.12) is solved iteratively by letting

$$T_{\psi_0} = \frac{W_\psi}{1 - \sum_{k=i}^{k=j} \frac{W_k}{\tau_{min}(e_k)}} \tag{5.18}$$

and then solving for T_{ψ_n} :

$$T_{\psi_n} = W_\psi + \sum_{k=i}^{k=j} \left\{ \left(\left\lceil \frac{T_{\psi_{n-1}}}{\tau_{min}(e_k)} \right\rceil W_k \right) \right\} \tag{5.19}$$

and stopping when $T_{\psi_n} = T_{\psi_{n-1}}$. The right-hand side is monotonically increasing with T_ψ and this process converges very rapidly since the initial guess is so near the final value.

Given a computation which takes a known time t in the absence of interruption, Algorithm 5.3, PTIME, computes the total time taken to do the computation in the presence of interrupts. It is assumed that there is no initiation delay involved, i.e. PTIME finds the worst case interval which contains t seconds of time in which preempting tasks are not executing.

Algorithm 5.3. PTIME(t, $E_{preempts}$)

Inputs: t, a time which represents computation time in the absence of preemption.

$E_{preempts}$, a set of events which can preempt the computation which takes t seconds.

Output: t_p, the time taken in the worst case with interrupts to perform a computation which takes t seconds without interrupts (i.e., PTIME assumes all the events in $E_{preempts}$ happen as soon as the computation starts, and continue at their maximum rate)

Method:

1. Let $W_{\downarrow}$ = t. Let $\{e_l, \cdots, e_j\}$ be the events in $E_{preempts}$. Let $\{W_l, \cdots, W_j\}$ be the weights of the subcontrol structures initiated by the corresponding events. Then solve equation (5.18) for an initial value of $T_{\downarrow}$; solve equation (5.19) repeatedly for $T_{\downarrow_n}$ using the value of $T_{\downarrow_{n-1}}$, ending when $T_{\downarrow_n} = T_{\downarrow_{n-1}}$. Return($T_{\downarrow_n}$).

5.4.4: Applications of PTIME

Using the algorithm PTIME one can determine several real-time properties of interest for control structures which meet the restrictions of Section 5.4. It must be kept in mind that there is a distinction between the following two sets of events:

a. The set of events which can preempt a task *after* it has been initiated, as well as take priority over its initiating event while it is pending.

b. The set of events which get priority over an event if it is pending but has not yet been recognized by the processor (no tasks have been initiated due to its occurrence), but cannot preempt any tasks in the subcontrol structure which that event initiates.

The worst case latency of any constraint which is in the subcontrol structure at priority 0 can also be directly determined. The distinction between this application and the one just mentioned is that the constraint need not be contained in a single copy of the subcontrol structure. Since the priority 0 subcontrol structure has no initiating event and hence no initiation delay, the worst case latency of a constraint C can be determined in two steps:

Algorithm 5.4. PRIOLATENCY(ψ, C)

Inputs: ψ, a subcontrol structure which runs at priority 0.

C, a constraint.

Output: l(C), the worst case latency of C in ψ.

Method:

1. Find (l(C), start_index, num_tasks) = ILATENCY(ψ, C), the worst case latency of C in the absence of preemption.

2. Let $E_{preempts}$ be the set of all events in the entire control structure. The worst case latency of C is PTIME(l(C), $E_{preempts}$).

Another application is to determine the latency of an event e_i, that is, how long is it in the worst case between the occurrence of an event and the initiation of the corresponding subcontrol structure. This can be found as follows:

Algorithm 5.5. ELATENCY(ϵ, e_i)

Inputs: ϵ, the least amount of time that can elapse before a task can be considered initiated.

e_i, the event whose latency is being determined.

Output: t_{e_i}, the longest time that can elapse after e_i occurs before its subcontrol structure gets initiated.

Method:

1. Let the set $E_{preempts}$ = $\{E_{always} \cup E_{win\_tie}\}$ relative to the event e_i.

2. t_{e_i} = PTIME(ϵ, $E_{preempts}$).

5.4.5: Adding Phase Relationships to PTIME

For a more general formulation, it is useful to have available the means of determining execution time in the presence of interruptions when the interrupting events may have started happening at any individually determined time rather than all starting at time zero. For this purpose, the *phase* of an event is here defined as the time since its last occurrence. Thus for a set of events $\{e_i, \cdots, e_j\}$ there may be associated a set of phases $\phi = \{\phi_i, \cdots, \phi_j\}$. If the events are occurring at their maximum rates, then no more than $\tau_{min}(e_i) - \phi_i$ seconds can elapse before the next occurrence of e_i.

In addition, there may be one or more pending occurrence of any of the events on the event queue, so a set of initially pending occurrences $\Omega = \{\Omega_i, \cdots, \Omega_j\}$ may be determined. These two factors alter the time due to preemption equation (5.12) as follows:

$$T_\psi = W_\psi + \sum_{k=i}^{k=j} \left(\left(\left\lceil \frac{T_\psi - \left(\tau_{min}(e_k) - \phi_k \right)}{\tau_{min}(e_k)} \right\rceil + \Omega_k \right) W_k \right) \tag{5.20}$$

A good lower bound to this is its solution without the ceiling function:

$$T_\psi \geq \frac{W_\psi + \sum_{k=i}^{k=j} \left[W_k \left(\Omega_k - \frac{\tau_{min}(e_k) - \phi_k}{\tau_{min}(e_k)} \right) \right]}{1 - \sum_{k=i}^{k=j} \frac{W_k}{\tau_{min}(e_k)}} \tag{5.21}$$

The solution is again found by solving (5.21) for the initial value T_{ψ_0} and then solving (5.20) for T_{ψ_n} using the previous value $T_{\psi_{n-1}}$ until they are equal. A summary is given below as Algorithm 5.6, PHTIME. Note that if $\phi_k = \tau_{min}(e_k)$ *and* $\Omega_k = 0$ for all *k*, PHTIME computes the same value as PTIME.

Algorithm 5.6. PHTIME(t, $E_{preempts}$, ϕ, Ω)

Inputs: t, a time which represents computation time in the absence of preemption.

$E_{preempts}$, a set of events which can preempt the computation taking t seconds.

ϕ, a set of phases, one for each event in $E_{preempts}$.

Ω, a set of initially pending occurrences, one for each event in $E_{preempts}$.

Output: t_{ph}, the time taken in the worst case to perform a computation which takes t seconds to perform with no interrupts. The worst case involves preemption by all the events in $E_{preempts}$ as often as possible, subject to the constraints of ϕ, Ω, and τ_{min} for each event.

Method:

1. Let W_ψ = t. Let $\{e_I, \cdots, e_J\}$ be the events in $E_{preempts}$. Let $\{W_I, \cdots, W_J\}$ be the weights of the subcontrol structures initiated by the corresponding events. Then solve equation (5.21) for an initial value T_{ψ_0}; solve equation (5.20) repeatedly for T_{ψ_n} using the previous value of $T_{\psi_{n-1}}$, terminating when they are equal. T_{ψ_n} is the value to be returned as t_{ph}.

5.4.6: Task Execution Time with Preemption at Priorities > 0

Algorithm 5.5 gives a method for determining the maximum time that can elapse between the occurrence of an event e_i and initiation of its subcontrol structure. This is fairly simply done since while e_i is pending the set of events that can preempt it is static. Once its subcontrol structure has been initiated, however, only events in E_{always} can interrupt; however, if any of these events does occur, any event in E_{win_tie} will take priority over resumption of e_i's subcontrol structure.

This complicates the determination of worst case execution time (and latencies, as will be seen in the next section) for a task subset $\mathcal{S}$ of the subcontrol structure. Note, however, that if the set E_{win_tie} is empty (and therefore the set of interrupting events is static), that PHTIME can be used to get the correct result.

In general though, the result must be found in stages, determining when $\mathcal{S}$ can be executed. The next algorithm determines the worst case time to execute a set of tasks $\mathcal{S}$, contained in a single subcontrol structure, given the sets of events E_{always} and E_{win_tie} for $\mathcal{S}$ and their initial values of ϕ and Ω. It assumes that $\mathcal{S}$ has been just initiated and then finds the time t_p from initiation to completion of $\mathcal{S}$. This is done by first finding how long it will be before all the pending interrupts, if any (based on ϕ and Ω), are processed and $\mathcal{S}$ can be resumed. Then the earliest occurrence of an event in E_{always} marks the next preemption of $\mathcal{S}$. At that point any accumulated occurrences of events in E_{win_tie} will cause executions of their subcontrol structures to be completed before $\mathcal{S}$ can be resumed. This partitioning of the total time taken to execute $\mathcal{S}$ is repeated until all of $\mathcal{S}$ is completed. Note that the method does not require determination of an exact schedule for all the tasks in the control structure, although the exact times when $\mathcal{S}$ will be executed are found. Algorithm 5.7, SCSTIME (for "subcontrol structure execution time") details the procedure. Note that this algorithm does not address the problem of determining execution time for a set of tasks which may require more than one invocation of a subcontrol structure.

Algorithm 5.7. SCSTIME($\mathcal{S}$, E_{always}, E_{win_tie}, ϕ, Ω)

Inputs: $\mathcal{S}$, a sublist of the tasks in a subcontrol structure.

E_{always}, relative to $e_{\mathcal{S}}$, $\mathcal{S}$'s initiating event.

E_{win_tie}, relative to $e_{\mathcal{S}}$.

ϕ, phases for events in E_{always} *and* E_{win_tie}.

Ω, initially pending occurrences for events in E_{always} *and* E_{win_tie}.

Output: t_p, the longest possible time to execute $\mathcal{S}$ with interruptions.

ϕ_{win_tie}, the final phases for all the events in E_{win_tie}.

Ω_{win_tie}, the final number of pending occurrences for all the events in E_{win_tie}.

Method:

1. Set $\mathcal{S}_{cum}$ = 0, the cumulative execution time for $\mathcal{S}$. Set t_1 = 0.

2. Find how long $\mathcal{S}$ can execute before it is preempted by an event from E_{always}. This is:

$$t_2 = \text{MINIMUM}\left(\tau_{min}(e_k) - \phi_k\right) \text{ for all } e_k \in E_{always} \tag{5.22}$$

Go to step (4).

3. Find how long δ can be executed before an event from E_{always} preempts it; this occurs at time:

$$t_2 = (\text{least multiple of } \pi_{min}(e_k) > t_1 \text{ for all } e_k \in E_{always}) \tag{5.23}$$

4. If $\delta_{cum} + t_2 - t_1 > |\beta|$, δ will complete in this interval; compute $t_p = t_1 + |\beta| - \delta_{cum}$; compute ϕ_{win_tie} using equation (5.25) and substituting t_p for t_2; compute Ω_{win_tie} using equation (5.24) and substituting t_p for t_2. Return $(t_p, \phi_{win\_tie}, \Omega_{win\_tie})$. Otherwise set $\delta_{cum} = \delta_{cum} + t_2 - t_1$.

5. Set $\Omega = 1$ for the event from E_{always} which caused the preemption. Some events in E_{win_tie} may also be pending:

$$\Omega_k = \left\lceil \frac{t_2}{\pi_{min}(e_k)} \right\rceil - \left\lceil \frac{t_1}{\pi_{min}(e_k)} \right\rceil \text{ for all } e_k \in E_{win\_tie} \tag{5.24}$$

6. Update phases for all events:

$$\phi_k = t_2 - \left\lfloor \frac{t_2}{\pi_{min}(e_k)} \right\rfloor \pi_{min}(e_k) \text{ for all } e_k \in \{E_{always} \cup E_{win\_tie}\} \tag{5.25}$$

7. Find new value of t_1, the next resumption time of δ:

$$t_1 = t_2 + \text{PHTIME}(\epsilon, E_{always} \cup E_{win\_tie}, \phi, \Omega) \tag{5.26}$$

8. Repeat steps (3) through (7) until termination of δ is detected in step (4).

5.4.7: Latencies for Constraints at Priorities > 0

The worst case latency may be desired for a constraint which is satisfied by an execution of a subcontrol structure at a priority greater than 0. If the execution which represents the greatest latency involves two or more invocations of that subcontrol structure, the possibility of initiation delay must be considered as well as interruption delay. Each of these delays may involve a different set of preempting events.

There are thus several complexities to be dealt with in the general case, even with control structures meeting the restrictions of Section 5.4; however there are also several special cases with simpler solutions. An example is when the sets E_{win_tie} and E_{lose_tie} are empty; it will be shown how to make use of this simplification in a later section.

Recall the notation of Section 5.2, where a subcontrol structure ψ was broken down into components $(\beta_1, \alpha_1, \cdots, \alpha_n, \beta_2)$ relative to a constraint C, where the α_i's were critical windows and the β_i's each contained one occurrence of C.

The worst case latency of C in a control structure containing ψ at a priority level greater than zero is found as follows. Let e_ψ be the initiating event for ψ. There are two candidate time intervals which may be the worst case latency for C. The first, t_{β_1}, is the maximum delay between occurrences of e_ψ plus the maximum delay to complete β_1 with preemption. The second, t_{α_m}, is the maximum time taken to complete α_m, the most critical window of ψ, also with preemption. Either one may involve more than one invocation of ψ, and hence initiation delay. To show that either t_{β_1} or t_{α_m} could be the worst case latency for C, consider a simple example:

Example 5.1. $((A^*/e1)B\ C\ D\ C)^*$.

where

$\tau_{max}(e1)$ = 10 sec.

$|A|$ = 1 sec.

$|B|$ = 2 sec.

$|C|$ = 1 sec.

$|D|$ = 3 sec.

The most critical window for the constraint (C) is (C D C), with a weight of 5 seconds. However, the longest time that elapses without an occurrence of C is 13 seconds, which is t_{β_1}, or $\tau_{max}(e1) + |B| + |C|$. If $|D|$ were changed to be 15 seconds, though, (C D C) would still be the most critical window for (C), but now t_{α_m} is 17 seconds, which is greater than t_{β_1}.

Thus the two candidate times must be computed and their maximum returned as $l(C)$. Note that since the entire control structure is repeated, the task list starting at β_2 and wrapping around through β_1 is a critical window, call it α_β, and must have weight greater than β_2; therefore β_2 cannot take longer than it to execute, and need not be considered as a candidate for $l(C)$. Furthermore, it might be thought that the weight of α_β plus the delay due to initiation of its second part, β_2, may in total be greater than the weight of an otherwise most critical window which is contained in ψ and hence has no initiation delay associated with it. To show this is untrue, it is only necessary to show that the weight of α_β with initiation delay must be less than t_{α_m} and t_{β_1}, since the addition of delays due to interruptions is a monotonically increasing function of the time taken without interruptions.

Thus assume that α_β is not the most critical window of ψ for C (if it is, it will be considered by the algorithms and thus there is no need to justify its exclusion). But if this is the case, then there is a critical window α_m in ψ with greater weight than α_β; thus the time to execute α_β is less than or equal to

$$|\alpha_\beta| + (\pi_{max}(e_\psi) - |\alpha_m|) \tag{5.22}$$

in the absence of interruptions. But since $|\alpha_\beta|$ is $\leq$ $|\alpha_m|$, equation (5.22) is $\leq$ $\pi_{max}(e_\psi)$. This in turn is less than t_β, which includes $\pi_{max}(e_\psi)$ as one of its summands. Thus it is sufficient to find the maximum of t_{β_1} and t_{α_m}.

Consider the computation of t_{α_m}. First the most critical window must be found for C in ψ using the algorithm for iterative control structures, ILATENCY. Note that in this case since the entire subcontrol structure gets repeated, the head (β_1) of $(\psi)^*$ containing C cannot represent the worst latency for C by itself (without initiation delay); there must be a critical window of greater weight which includes β_1 as its second occurrence of C.

Therefore ILATENCY will return l(C), the weight of the most critical window α_m

in $(\psi)^*$. ILATENCY also returns start_index, the index in ψ of the first task of α_m, and num_tasks, the number of tasks in α_m. Knowing this, it can be determined how many times e_ψ, the initiating event for ψ, must occur during α_m's execution (i.e., by knowing how many copies of ψ are included in α_m). Partition α_m into the sublists $\{\alpha_{m_1}, \alpha_{m_2}, \cdots, \alpha_{m_n}\}$, where each α_{m_i} is a portion of α_m which is contained in (a single copy of) ψ. Since t_{α_m} is the longest possible time to execute α_m, it must be assumed that all the interrupts happen immediately after initiation of α_m and continue at their maximum rates, while the initiating event e_ψ happens at its slowest rate.

Figure 5.4 shows the time line for part of a sample execution of a critical window α_m which is not contained by a single copy of ψ.

```
|----1----|----2----|----3----|----4----|----5----|----6----|
e         ψ         α         α         e         α
ψ         starts    m         m         ψ         m
occurs               1         1        occurs     2
                    starts    ends      second    starts
                                        time
```

Fig. 5.4. Partial execution of a critical window α_m.

In the worst case, the initiation delay of interval (4) will be the maximum possible, with the constraint that interval (3) must be at its maximum too (greatest amount of time lost to interrupts). Therefore the intervals (1) and (2) must be computed at their minimum, i.e. no preemption. Thus interval (1) is assumed to be zero, and interval (2) is $|\psi| - |\alpha_{m_1}|$. This may give an inflated upper bound by lengthening interval (4); if it is known in a particular case that preemption must occur during intervals (1) and (2), an adjustment can be made in the phases of the interrupting events at the beginning of interval (3).

As was previously stated, it is assumed that the worst case is when all events occur right after α_{m_1} starts, so the length of interval (3), $t_{(3)}$, is found from SCSTIME(α_{m_1}, E_{always}, E_{win_tie}, ϕ, Ω) where E_{always} and E_{win_tie} are determined relative to e_ψ, $\phi = (0, \ldots, 0)$ and $\Omega = (1, \ldots, 1)$ for all the events.

Once the interval times $t_{(1)}$, $t_{(2)}$, *and* $t_{(3)}$ are determined, $t_{(4)}$ is found by:

$$t_{(4)} = \text{MAXIMUM}\left[0, \tau_{max}(e_\psi) - t_{(1)} + t_{(2)} + t_{(3)}\right] \qquad (5.27)$$

If $t_{(4)} > 0$, there is an initiation delay which must be factored into the solution.

At this point another decision must be made which affects the tightness of the upper bound determined by the algorithm. During interval (4), any of the events in the control structure other than e_ψ may get control, and there may be arbitrarily complex blocking out among the different sets of events due to the exact order of occurrences; i.e., to get the true picture, the sets E_{always}, E_{win_tie} and E_{lose_tie} relative to *every* event must be considered, since the reference point provided by knowledge that e_ψ was pending has been lost. This makes finding an analytic solution for the values of ϕ and Ω at the end of interval (4) quite complicated, and two alternatives are provided here instead. Note that the relative importance of this is dependent on the relative size of interval (4); in the extreme case, if it is zero, then there is no problem at all.

The simpler method (and the one used here) is to assume that all events in E_{always} and E_{win_tie} get blocked out during interval (4), and thus their ϕ's and Ω's get updated accordingly. This will provide an upper bound which is high by the amount of execution of preempting tasks which could have taken place during interval (4) and will now instead be added to the preemption delays of the next interval.

Unfortunately, this is not the only complication. In the worst case, an event from E_{lose_tie} might get control just before the end of interval (4), and initiate a subcontrol structure which could not be preempted by e_{ϕ}. The event e_l in E_{lose_tie} which initiates a subcontrol structure that runs for the longest time without being preempted by an event in E_{always} or E_{win_tie} (given their ϕ's and Ω's at the end of interval (4)) is chosen, since once it gets preempted it has less priority than e_{ϕ} by definition. Let the length of this time be t_l, and then the time until α_{m_2} starts is given by PHTIME(t_l, $\{E_{always} \cup E_{win\_tie}\}$, ϕ, Ω). The ϕ's and Ω's are updated and the process is repeated as from the start of α_{m_1}, terminating when the end of α_{m_n} is reached.

The alternative method is to determine an exact schedule for interval (4). Then it will be known whether or not an event from E_{lose_tie} can get control and keep it past the end of interval (4), and the exact ψ's and Ω's for all the events can be determined. This is the method of choice if the initiation delay is known to be significant.

The interval t_{β_1} is measured on a slightly different time line:

```
|----0----|----1----|----2----|----3----|----4----|----5----|
          e         β         β         e         β
          ψ         1         1         ψ         1
          occurs     1         1        occurs     2
                    starts    ends      second    starts
                                        time
```

Fig. 5.5. Partial execution of β_1.

To find t_{β_1}, the execution of β_1 is broken down into parts which are contained in a single copy of ψ, just as was done for α_m. Here the worst case is when all interrupts happen at the beginning of interval (1) and continue at their maximum rate, since the length of interval (0) is fixed at $\pi_{max}(e_\psi)$; this gives the greatest delay during interval (1). Interval (1) is thus the maximum initiation delay for ψ with preemption, including the possibility of an event from E_{lose_tie} getting control just before e_ψ happens and causing further delay as previously discussed. The times of the remaining intervals are found as was done for the α_{m_i}'s, computing the initial ψ's and Ω's appropriately.

This procedure is detailed in Algorithm 5.8, LATENCY.

Algorithm 5.8: LATENCY(C, ψ)

Inputs: C, a constraint

ψ, a subcontrol structure containing all the tasks in C, in a control structure meeting the restrictions of Section 5.4, and where the worst case latency of C is known not to be infinite by equation (5.11).

Output: l(C), the worst case latency of C in the control structure containing ψ.

Method:

1. Find l(C), start_index, and num_tasks by executing ILATENCY((ψ)*, C). Let ω_m be the critical window starting at start_index and continuing for num_tasks.

2. Find the sublists of ω_m: $(\omega_{m_1}, \omega_{m_2}, \cdots, \omega_{m_n})$ where each ω_{m_i} is the completely contained in a single copy of ψ. If the number of tasks in ψ is k, then ω_{m_1} = ψ[start_index] through $\psi[k]$, ω_{m_2} through $\omega_{m_{n-1}} = \psi$, and ω_{m_n} = ψ[1] through ψ[num_tasks - $k(n$ - 2) - (k - start_index + 1)].

3. Since the worst case involves maximum initiation delay for ψ, assume intervals (1) and (2) (see Figure 5.4) elapse without preemption. Thus $t_{(1)} = 0$ and $t_{(2)} = |\psi| - |\omega_{m_1}|$, and $\phi_\psi = t_{(1)} + t_{(2)}$, at the start of interval (3).

4. Find the sets E_{always}, and E_{win_tie} relative to e_ψ. Set $\phi = 0$ and $\Omega = 1$ for all events in these sets. Find the set E_{lose_tie} relative to e_ψ. Set $t_{\omega_m} = 0$. Initialize the counter $i = 0$. Repeat steps (5) through (7) until the end of ω_{m_n} is reached in step (5).

5. Set $i = i + 1$. Find $t_{(3)} = t_p$, which is returned by SCSTIME(ω_{m_i}, E_{always}, E_{win_tie}, ϕ, Ω). Set $t_{\omega_m} = t_{\omega_m} + t_{(3)}$. Set ϕ and Ω for the events in E_{win_tie} to the values ϕ_{win_tie} and Ω_{win_tie} returned by SCSTIME. If $i = n$, go to step (8) where t_{ϕ_1} is computed.

6. At the end of $t_{(3)}$, since ω_{m_i} was in control, none of the events in E_{always} was pending. Thus set $\Omega = 0$ and:

$$\phi_k = t_{e_m} - \left\lfloor \frac{t_{e_m}}{\tau_{min}(e_k)} \right\rfloor \tau_{min}(e_k) \text{ for each event } e_k \in \{E_{always}\} \tag{5.28}$$

7. Let $t_{(4)} = \tau_{max}(e_\psi) - t_{(3)} - \phi_\psi$. If $t_{(4)} > 0$, there is an initiation delay and the following must be done:

a. Update ϕ and Ω for each event e_k in $\{E_{always} \cup E_{win\_tie}\}$:

$$\Omega_k = \left\lfloor \frac{t_{(4)} + \phi_k}{\tau_{min}(e_k)} \right\rfloor \tag{5.29}$$

$$\phi_k = t_{(4)} + \phi_k - \Omega_k \, \tau_{min}(e_k) \tag{5.30}$$

b. Find the event $e_i \in E_{lose\_tie}$ which initiates a subcontrol structure that can run the longest before (or without) being preempted by an event in $\{E_{always} \cup E_{win\_tie}\}$; this can be done by considering each event in E_{lose_tie} in turn. Let t_{e_i} be the time which elapses past the end of interval (4) due to e_i.

c. Find the initiation delay of $e_{m_{i+1}}$:

$$t_{delay} = \text{PHTIME}(t_{e_i}, \{E_{always} \cup E_{win\_tie}\}, \phi, \Omega) \tag{5.31}$$

d. Set $t_{e_m} = t_{e_m} + t_{(4)} + t_{delay}$.

e. Set $\Omega_k = 0$, and:

$$\phi_k = t_{e_m} - \left\lfloor \frac{t_{e_m}}{\tau_{min}(e_k)} \right\rfloor \tau_{min}(e_k) \tag{5.32}$$

for all events e_k in $\{E_{always} \cup E_{win\_tie}\}$.

e. Set $\phi_\psi = t_{delay}$.

If $t_{(4)}$ is zero, set $\phi_\psi = \phi_\psi + t_{(3)} - \tau_{max}(e_\psi)$.

8. Find t_{β_1}; find β_1 of $(\psi)^*$ by scanning until the first occurrence of C has been scanned. Divide β_1 into sublists as was done for α_m in step (2), getting as a result $(\beta_{1_1}, \beta_{1_2}, \cdots, \beta_{1_n})$, where this n may be different from the n obtained for α_m.

9. Refer to Figure 5.5. The time of interval (0), $t_{(0)}$, is $\tau_{max}(e_{\psi})$. Assume all events in $\{E_{always} \cup E_{win\_tie}\}$ occur at the end of this interval, and continue at their maximum respective rates. Thus set $\Omega = 1$ and $\phi = 0$ for all these events. Let $T_{\beta_1} = t_{(0)}$; let $i = 0$. Starting at step (7b), execute just as for α_m, substituting t_{β_1} for t_{α_m}, and β_{1_i} for α_{m_i}.

10. Return MAXIMUM$(t_{\beta_1}, t_{\alpha_m})$.

5.5: Special Cases and Extensions

There are many special cases which result in much simpler algorithms. Each algorithm presented in the previous section is directed towards a subset of control structure types which contains the previous subset and some additional control structure types; it is seen that in general, as the number of different types in the subset increases, so does the complexity of the resulting algorithms.

As an example of another important special case, consider finding any of the real-time properties for a subcontrol structure whose sets E_{lose_tie} and E_{win_tie} are empty, e.g., as would be the case in a control structure containing no event coupled lists. Now all of the complications due to having the set of preempting event variables change dynamically drop out -- the statically determined set E_{always} is the only set that may preempt, and by definition it can always preempt.

The simplifications this introduces are substantial; take the most complex of the algorithms of the previous section, Algorithm 5.8, LATENCY, for example. In step (5), SCSTIME can be replaced by the simpler PHTIME. There may still be an initiation delay $t_{(4)}$, but there is no longer the possibility of an event from E_{lose_tie} getting control and prolonging the initiation time.

As far as extensions to the algorithms go, there are two principal areas to consider: one is the determination of algorithms for real-time properties not discussed here and which are germane to a specific application, and the other is the lifting of the restrictions of Section 5.4 to allow any representable control structure to be analyzed. Since the first area requires an application relative to which suitable algorithms can be developed, only the second area will be covered here.

The difficulty involved in lifting the restrictions of Section 5.4 varies considerably from one restriction to the next, and hence they are discussed here one at time. The following discussions are not intended to be the final word on the topic, nor are all the details supplied for a particular method of lifting each restriction. Instead, the intention is to point out the difficulties involved in each case and to make suggestions as to how they might be overcome.

5.5.1: External Termination

Recall that there are two types of iteration, in effect, that can be applied to a subcontrol structure; local and global. If a subcontrol structure is locally cyclic, it means that that particular subcontrol structure executes indefinitely, without requiring reinitiation by its initiating event. This is equivalent, then, to having an event which initiates a subcontrol structure with infinite weight. If, instead, it is part of a globally cyclic control structure, then it too will be repeated indefinitely, but only one time per initiating event occurrence. Both of these types are allowed under the restrictions of Section 5.4, because the weights of the initiated subcontrol structures are fixed, even though they may be infinite in the locally cyclic case. However, there is the potential for a subcontrol structure which has infinite (and thus fixed) weight with no external termination to have varying weight in the presence of external termination. Thus the delays encountered in the execution of lower priority control structures due to interrupts which initiated <abort cs>'s (those which may be externally terminated) will vary according to how long the <abort cs> executes before *it* gets preempted. An upper bound on this time can be found if a good value is known for π_{max} of the terminating event; if there is more than one such event, the minimum of their maximum periods may be used.

Note that this also complicates the determination of load factor (equation (5.11)), since that depends as well on having a known upper bound for the weight of each subcontrol structure.

5.5.2: Restart Control Structures

This is another case which may lead to variable subcontrol structure execution times. Every time a <restart cs> gets preempted, the time of its current execution is extended by its nominal weight in the absence of preemption; it is essentially the opposite of external termination. Thus a <restart cs> needs a non-preempted interval equal to its nominal weight in which to execute. To find whether such an interval exists, one must see whether the phases of all the events in the sets E_{always} and E_{win_tie} relative to the <restart cs> can be adjusted so that it gets preempted at least once every |<restart cs>| - ϵ seconds. This can be either very simple, as in the case where there is only one event that can preempt the <restart cs>, or very complex, if there are many events and their interrelationships must be considered.

5.5.3: Codestripping

This is somewhat simpler to handle. If one of the interrupting events initiates a <codestripped cs>, then the delay it causes is simply its nominal weight divided by the number of codestrips, e.g. the weight of (A/5) is just |A|/5. If the tasks whose execution time is being measured are codestripped, though, it is as if they were preempted by an event with variable π_{min} -- to get this effect, a dummy event can be substituted for the integer which tells how many codestrips there are, and its phase can be adjusted every time the <codestripped cs> is resumed so that it will cause preemption at the time when a single codestrip would have finished.

5.5.4: Non-Preemptible Tasks

Let ψ_{meas} be a subcontrol structure whose real-time properties are being measured. Then if a subcontrol structure of higher priority than ψ_{meas} includes non-preemptible tasks, the effect on ψ_{meas} is unnoticeable -- these tasks would have been executed to completion anyway before ψ_{meas} was resumed. If all of ψ_{meas} is non-preemptible, then its computation time need not include the effects of those interrupts which cannot preempt it, and the sets E_{always} and E_{win_tie} can be adjusted accordingly. If only a part of ψ_{meas} is non-preemptible, then the ψ's and Ω's of interrupting events must be updated when the non-preemptible part has been executed. If a subcontrol structure of lower priority than ψ_{meas} is non-preemptible, then if the interval ψ_{meas} includes an initiation delay, it must be increased by the maximum amount possible due to execution of tasks which e_ψ cannot preempt. This can be handled similarly to the case where an event from E_{lose_tie} gets control just before e_ψ occurs.

5.5.5: Stopping the Flow of Control

This is another case which may result in effectively varying the weights of subcontrol structures and hence the delay due to preemptions which include their execution. It has some similarities to external termination; consider the example given in equation (5.7), repeated here:

$$((((A^*/e1)B)/e2)C)^*$$

The problem is that the effect of the delay in executing A due to e1's occurrence is dependent on the period of e2 -- hence the similarity to external termination. The difference is that the minimum effective weight of B is still |B|, since an occurrence of e2 before the end of B preempts B, but leaves the remainder of B to be resumed once C is done.

Thus the techniques for external termination can be applied here, with the constraint that the minimum weight of a subcontrol structure is still its nominal weight.

5.5.6: Constraints at More than One Priority Level

To be able to consider the worst case latencies of constraints whose member tasks are found at different priority levels and thus in different subcontrol structures is a difficult problem. To determine this, the executions of tasks at lower and higher priority levels can no longer be lumped together and treated as a delay, since at the very least it must be known when every task which occurs in the constraint is executed, regardless of what its priority may be. Thus algorithms of a very different sort from those in the previous sections are probably required, and the possibility of simulation to determine an exact schedule may provide a starting point.

5.5.7: Finite Event Queues

If only a finite number of event occurrences can be remembered, and this number is small enough so that some event occurrences are ignored, then from ψ_{meas}'s point of view, the delays due to preemption computed previously may be too high but cannot be too low. The equations which determine the time lost to preemption must be adjusted to include a maximum value of Ω.

When computing initiation delay, it must now be seen whether, in the worst case, the initiation delay may be prolonged due the initiating event's occurrence being ignored.

6: Conclusions and Directions for Future Research

A new notation has been given which represents real-time control structures at a high (task and event) and implementation-free level, including sequencing, iteration and preemption as primary constructs. The notation can represent conventional single and multiple level interrupt structures as well as non-traditional ones where branching of the preemption structure is generalized. A total priority ordering may be described, or arbitrarily many events and subcontrol structures may reside at the same priority level. An algorithm is given for determining the preemption relationship for any <event, task> couple in the control structure, as well as a completely deterministic method of selecting a task for service if several events with arbitrary priorities are pending (possibly equal). It may be interesting to consider the modifications necessary to the algorithms if it is assumed that the processor chooses at random from among all the pending events of the highest priority.

Additionally, notation is given for representing task termination by external event occurrences (as opposed to temporary preemption), describing whether a control structure should be restarted from its first task or resumed from the point of preemption, codestripping, and masking of a set of interrupts while any given task is executing. It is shown that due to the assumed transitivity of the "preempts" relation, the sets of events chosen for these special cases might necessarily include other events not explicitly mentioned.

The notation is compact, and provides a convenient format for conveying a lot of information about the control flow relationships among the members of a set of tasks. A complete BNF specification is provided, and a parser can be (and has been) constructed using any of a number of extant compiler-compilers which accept BNF specifications.

Classes of representable control structures are given, typed by the topology of their control flow graphs. It is shown that partial as well as total orderings of tasks and events can be achieved through the use of the event coupled list, which introduces forks into the control flow graph. A method for recursively constructing a multiple priority level control structure of the traditional type is given. The distinction is made between a control structure which supports a processor priority and one which actually has only a single level of interrupts, even though there may be a set of several interrupting events which are ordered among themselves. It is shown that while in general the need for this type of control structure is perceived to be strongest in situations where representation of periodic events and task executions prevails, aperiodic control structures are representable. However, a true tree-shaped interrupt structure cannot be achieved due to the transitivity of the "preempts" relation. In addition, while iteration can be applied to any closed or basic control structure, a back arc cannot originate from the middle of one event coupled list and terminate in the middle of another. This is not felt to be a serious restriction, however, since it is likely that groups of tasks in a subcontrol structure are related and expected to be executed as a block.

The second half of the thesis concentrates on describing the sorts of real-time properties which may be of interest to a user of any real-time system, and demonstrating how they can be measured for control structures representable using the notation presented here. The worst case latency of a constraint is found to be a property whose determination involves computation of several other properties as subroutines. The difficulty of finding an upper bound on task execution time is discussed, although without this knowledge it is doubtful that much further analysis of value could be performed. Additionally, bounds on the maximum and minimum period for each event are needed. The algorithms reflect reality in that if these periods are not known, it will be difficult to forecast real-time performance for the control structure.

Next several algorithms for measuring latencies are developed, each handling a larger set of control structure types, up to a level which includes the entire basic framework of sequencing, iteration and preemption. Along the way, it is shown how to determine if a response time might be infinite, and it is assumed that this is done before attempting to use any of the algorithms for measuring the various time intervals. An algorithm is given which determines the loss of time due to preemption if the set of preempting events is static, and by using it it is shown how to determine the latency of a constraint contained in a priority 0 subcontrol structure, and the worst case initiation delay for an event at a given priority level. The worst case assumed here is the occurrence at the beginning of an interval of all interrupts, and their reoccurrence at their individual maximum rates. However, an algorithm is also given which determines preemption time if the phase of each event is known at the beginning of the interval being measured.

The effects on these algorithms of adding control structures containing each of the restricted items of Section 5.4 is considered; further investigation is needed here to uncover the details of the problems which are pointed out. Another useful thing would be to develop analyses based on a probabilistic model rather than on the worst case; e.g., what is the probability that a given constraint will have a latency of no more than *n* seconds? Finally, an important result would be the development of a general algorithm which could determine the latency for any of the representable control structures. The difficulty of such a task should not be underestimated; indeed, in the words of Niklaus Wirth:

> It does not appear feasible at this time to postulate any generally valid and at the same time practically useful rules for the determination of execution time bounds for systems using processor sharing. [Wirth 77b]

Appendix A: Summary of BNF for Real-time Control Structures

```
<control structure> ::= <basic cs> | <closed cs> | <iterative cs>

<task id> ::= <letter> | <task id> <alphanumeric>

<letter> ::= A | B | C | ... | Z

<alphanumeric> ::= <letter> | <digit>

<digit> ::= 0 | 1 | 2 | ... | 9

<basic cs> ::= <task> | <basic cs> ↓̸ <task> | <basic cs> ↑

<task> ::= <task id> | <non-preemptible tid> | <abort tid>

<closed cs> ::= ( <basic cs> ) | ( <preemptible cs> ) | ( <closed cs list> ) |

        ( <closed cs> <preemptible cs> ) | ( <closed cs> <basic cs> ) |

        ( <restart cs> ) | <non-preemptible closed cs> | <abort cs>

<closed cs list> ::= <closed cs> | <closed cs list> <closed cs>

<iterative cs> ::= <basic cs>* | <closed cs>* | <basic cs> <iterative cs>

<preemptibie cs> ::= <control structure> / <event list> | <codestripped cs>

<event var> ::= e<integer>

<integer> ::= <digit> | <integer> <digit>
```

```
<event list> ::= <event var> | (<event coupled list>) |

        (<event coupled list>)*

<event coupled list> ::= <event var>: <control structure> |

        <event coupled list> '|' <event var>: <control structure>

<non-preemptible tid> ::= '<task> | '(<ev list>)<task>

<non-preemptible closed cs> ::= '<closed cs> | '(<ev list>)<closed cs>

<ev list> ::= <event var> | <ev list>,<event var>

<abort tid> ::= @<task> | @(<ev list>)<task>

<abort cs> ::= @<closed cs> | @(<ev list>)<closed cs>

<restart cs> ::= > <basic cs> | > (<ev list>) <basic cs>

<codestripped cs> ::= <basic cs> / <integer>
```

REFERENCES

[Benson 67] Benson, D., R.J. Cunningham, I.F. Currie, M.R. Griffith, R. Kingslake, R.J. Long, and A.J. Southgate, "A language for real-time systems," *The Computer Bulletin* 11,3 (Dec. 1967), 202-212.

[Dijkstra 68] Dijkstra, E.W., "Cooperating sequential processes," in *Programming Languages* (F. Genuys ed.), Academic Press, NY, 1968, 43-112.

[Dijkstra 72] Dijkstra, E.W., "A class of allocation strategies inducing bounded delays only," *AFIPS Conf. Proc.* 40 (1972 SJCC), 933-936.

[Fosdick 76] Fosdick, L.D., and L.J. Osterweil, "Data flow analysis in software reliability," *Computing Surveys* 8,3 (Sept. 1976), 305-330.

[Freiburghouse 77] Freiburghouse, R.A., "Proposed extensions to PL/I for real-time applications," *SIGPLAN Notices* 12,7 (July 1977), 26-42.

[Gonzalez 77] Gonzalez, M.J. Jr., "Deterministic processor scheduling," *Computing Surveys* 9,3 (Sept. 1977), 173-204.

[Hennessy 75] Hennessy, J.L., R.B. Kieburtz, and D.R. Smith, "TOMAL: A task-oriented microprocessor applications language," *IEEE Transactions Ind. Elect. Cont. Inst.* IECI-22,3 (Aug. 1975), 283-289.

[Hoare 74] Hoare, C.A.R., "Monitors: An operating system structuring concept," *Comm. ACM* 17,10 (Oct. 1974), 549-557.

[Kieburtz 75] Kieburtz, R.B., and J.L. Hennessy, "TOMAL - A high level programming language for microprocessor process control applications," *Proc. ACM SIGMINI/SIGPLAN Interface Meeting on Prog. Systs. in a Small Processor Environment*, also *SIGPLAN Notices* 11,4 (April 1976), 127-133.

[Liu 73] Liu, C.L., and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM* 20,1 (Jan. 1973), 46-61.

[Ormicki 77] Ormicki, A., "Real-time BASIC for laboratory use," *Software Prac. & Exp.* 7,4 (July-Aug. 1977), 435-444.

[Phillips 76] Phillips, J.V., and T.H. Bredt, "Design and verification of real-time systems," *Proc. IEEE 2nd Int. Conf. on Soft. Eng.* (Oct. 1976), 124-131.

[Schoeffler 70] Schoeffler, J.D., and R.H. Temple, "A real-time language for process control," *Proc. of IEEE* 58,1 (Jan. 1970), 98-110.

[Serlin 72] Serlin, O., "Scheduling of time critical processes," *AFIPS Conf. Proc.* 40 (1972 SJCC), 925-932.

[Teixeira 78] Teixeira, T.J., *Real-time control structures for block diagram schemata*, S.M. Thesis, Department of Electrical Engineering and Computer Science, M.I.T., January 1978.

[Wirth 77a] Wirth, N., "Modula: A language for modular multiprogramming," *Software Prac. & Exp.* 7,1 (Jan.-Feb. 1977), 3-35.

[Wirth 77b] Wirth, N., "Toward a discipline of real-time programming," *Comm. ACM* 20,8 (Aug. 1977), 577-583.